Hopscotch: A Hardware-Software Co-Design for Efficient Cache Resizing on Multi-Core SoCs

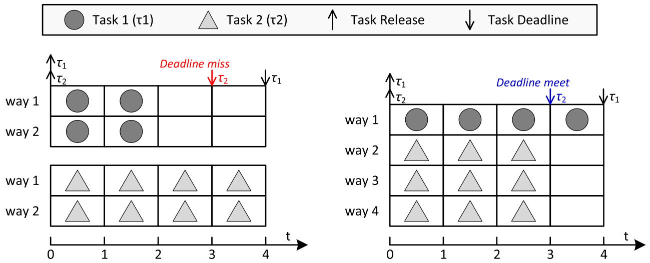
Zhe Jiang, *Member, IEEE*, Kecheng Yang, Nathan Fisher, Nan Guan, *Member, IEEE*, Neil C. Audsley, and Zheng Dong

***Abstract*—Following the trend of increasing autonomy in real-time systems, multi-core System-on-Chips (SoCs) have enabled devices to better handle the large streams of data and intensive computation required by such autonomous systems. In modern multi-core SoCs, each L1 cache is designed to be tied to an individual processor, and a processor can only access its own L1 cache. This design method ensures the system's average throughput, but also limits the possibility of parallelism, significantly reducing the system's real-time schedulability. To overcome this problem, we present a new system framework for highly-parallel multi-core systems, *Hopscotch*. *Hopscotch* introduces re-sizable L1 cache which is shared between processors in the same computing cluster. At execution, *Hopscotch* dynamically allocates L1 cache capacity to the tasks executed by the processors, unblocking the available parallelism in the system. Based on the new hardware architecture, we also present a new theoretical model and schedulability analysis providing cache size selection methods and corresponding timing guarantees for the system. As demonstrated in the evaluations, *Hopscotch* effectively improves system-level schedulability with negligible extra overhead.**

***Index Terms*—Real-time systems, hardware/software co-design, L1 cache, schedulability analysis.**

I. Introduction

A MAJOR factor in the recent trend towards increasingly autonomous systems is the proliferation of relatively inexpensive, yet highly-parallel multi-core System-on-Chips (SoCs). These parallel embedded SoCs have enabled devices to better handle the large streams of data and intensive computation required to learn and make decisions autonomously in uncertain, high-dimensional environments, using techniques like deep learning. However, while the explosion of highly-parallel platforms has seen a proportionate growth in the number of applications/devices that use these platforms, understanding in the embedded systems community of how to build time-predictable, safety-critical systems with such parallel architectures has not kept pace, especially in the L1 cache.



(a) Isolated L1-cache

(b) Shared and resizable L1-cache

Fig. 1. Task scheduling with isolated L1-cache (i.e., each processor has two designated cache ways), and shared and re-sizable L1-cache (t: time; way: cache way).

L1 cache is a vital resource in multi-core SoCs, buffering the contents stored in memory and providing a fast path for processor access. With the L1 cache, the processor can obtain the desired data/instruction within one or two clock cycles, rather than wasting dozens of cycles waiting for data/instructions to return from memory or low-level cache. Hence, the effectiveness of using L1 cache is a dominant factor when determining the utilization, throughput, and real-time schedulability of the entire system [25].

Conventionally, L1 cache is designed to be tied to an individual processor, and a processor is only allowed to access its designated L1 cache. This design approach provides fixed cache capacity for each processor, ensuring the processors' average throughput [38]. However, isolated cache partitioning also limits available parallelism in the system, reducing the system's real-time schedulability. For example, Fig. 1(a) shows two tasks τ_1 and τ_2 deployed in a dual-core system. At time point 0, τ_1 and τ_2 are released and executed by the processors simultaneously. At time point 2, τ_1 completes the execution, and its L1 cache becomes free. However, due to the restriction, τ_2 can still only execute with the limited cache capacity until time point 4, missing the deadline.

To effectively exploit the available parallelism and ensure the system's real-time schedulability, L1 cache must not become the bottleneck. Therefore, instead of isolating the L1 cache, we can share the L1 cache between certain processors and dynamically

Manuscript received 1 April 2023; revised 26 August 2023; accepted 10 November 2023. Date of publication 14 November 2023; date of current version 27 November 2023. This work was supported in part by the U.S. National Science Foundation under Grants CNS-2103604, CNS-2140346, CNS-2231523, IIS-1724227, CNS-2038609, CNS-2211641, and CNS-2104181. Recommended for acceptance by S. Song. *(Corresponding author: Zheng Dong.)*

Zhe Jiang is with the School of Integrated Circuits, Southeast University, Nanjing 21000, China, and also with Computer Science Department, University of Cambridge, CB3 0FD Cambridge, U.K. (e-mail: zhejiang.uk@gmail.com).

Kecheng Yang is with the Department of Computer Science, Texas State University, San Marcos, TX 78666 USA (e-mail: yangk@txstate.edu).

Nathan Fisher and Zheng Dong are with the Department of Computer Science, Wayne State University, Detroit, MI 48202 USA (e-mail: fishern@wayne.edu; dong@wayne.edu).

Nan Guan is with Computer Science Department, City University of Hong Kong, Kowloon Tong, Hong Kong (e-mail: nanguan@cityu.edu.hk).

Neil C. Audsley is with the Department of Computer Science, City, University of London, EC1V 0HB London, U.K. (e-mail: neil.audsley@york.ac.uk).

Digital Object Identifier 10.1109/TPDS.2023.3332711

allocate the cache capacity to each processor, adjusting task execution time to guarantee overall schedulability. Consider the same example discussed above, with shared and re-sizable L1 cache (Fig. 1(b)). More cache capacity can be allocated to τ_2 by "borrowing" the cache ways from τ_1, accelerating τ_2's execution. Although this adjustment slows down τ_1's execution, it ensures system-level schedulability.

Contributions: With this in mind, we present a new system framework (*Hopscotch*) for highly-parallel multi-core systems. Unlike conventional multi-core systems, *Hopscotch* introduces novel re-sizable L1 cache (*Hopscotch-Cache*), shared between processors in the same computing cluster. We also introduce a new analysis framework to examine the system's schedulability. With that, we present a configuration algorithm to dynamically allocate cache capacity to the tasks executed by the processors, unblocking the available parallelism and ensuring overall real-time schedulability. The contributions are summarized as follows:

- At the hardware level, a novel *micro-architecture* for L1 cache is developed. This design supports partial cache sharing and run-time resizing between processors without causing any extra critical paths.
- At the software level, new *analysis frameworks* and a *selection algorithm* are presented. The analysis frameworks provide a theoretical evaluation of schedulability yielded by the new L1 cache, and the selection algorithm allocates the cache capacity for the processors, thereby enabling the available parallelism.
- As a full-stack solution, a complete *system architecture* is introduced, effectively integrating the new hardware and software to construct the system, guaranteeing real-time schedulability with improved throughput.
- Comprehensive evaluation is proceeded to examine the new systems in terms of overhead, performance, effectiveness, and scalability.

The rest of the paper is organized as follows: Section II presents the motivation and research challenges, followed by the design of *Hopscotch* in Sections III and IV. Section V presents the theoretical analysis and cache size selection methods for *Hopscotch*. Sections VI and VII evaluate the overhead and real-time performance of *Hopscotch*, respectively. Section VIII reviews the related work and Section IX concludes the paper.

II. Motivation and Research Challanges

In this section, we first explain the motivation for implementing a new real-time system with shared and re-sizable L1 cache, and then present the research challenges.

A. Execution Through a Pipelined Processor

Instructions are a processor's basic processing objects, and a software task usually comprises thousands of instructions. Therefore, we briefly review how instructions are processed in a modern pipelined processor with L1 cache, and examine how each stage contributes to the overall execution time. Fig. 2 illustrates the top-level micro-architecture of a processor, executing each instruction in 5 pipeline stages [25]:

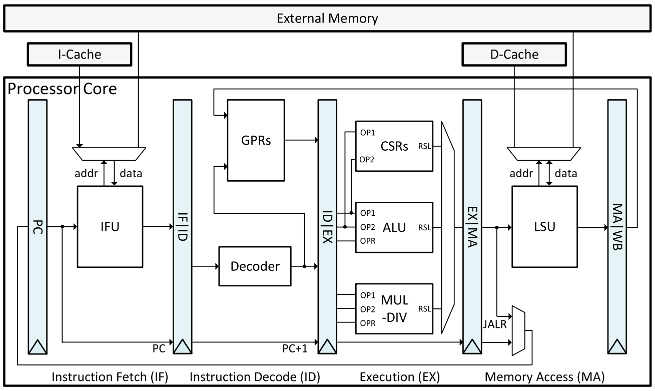


Fig. 2. Top-level micro-architecture of a 5-stage RISC-V processor (OP: Operand; OPR: Operator; RSL: Result; CSR: Control Status Register; MUL: Multiplier; Div: Divider). Grey shading indicates cache and external memory.

- *Instruction Fetch (IF):* the processor fetches an instruction from its instruction cache (I-cache, if cache hits) or external memory (DDR, if cache misses) using an Instruction Fetch Unit (IFU).
- *Instruction Decode (ID):* the processor decodes the fetched instruction and then stores the decoded results in its General-Purpose Registers (GPRs).
- *Execution (EX):* the processor executes the decoded instructions by operating corresponding execution units. For example, an Arithmetic Logical Unit (ALU) calculates the addition or subtraction of integers.
- *Memory Access (MA):* the processor reads/writes data from/to its data cache (D-cache, if cache hits) or memory (if cache misses) using a Load Store Unit (LSU).
- *Write-Back (WB):* the processor stores the results back to GPRs, and starts the next-round of execution.

Execution Time of Each Processing Stage: To understand the relationship between the processing stages and instruction execution time, we run PARSEC benchmarks (v3.0) [9], except raytrace, vips, dedup, and canneal,[1] with the simsmall input, using a RISC-V processor with L1 cache on our experimental platform (Xilinx VC707 using the configuration given in Section VI). In the experiments, we connected a cycle-accurate counter to the processor, driven by the same clock source as the processor, measuring the clock cycles taken by instruction executions at each stage.[2]

As shown in Fig. 3, the IF and MA with cache-hit (CH), and the ID and WB usually completed in one or two clock cycles. The EX had a variable execution time due to the calculations. The most significant time overhead was observed at IF and MA with *cache-miss* (CM): the processor required more than 15 clock cycles to read/write the corresponding content to/from the external memory, which *dominated* the execution time of the entire instruction. Note that, since Xilinx VC707 (an FPGA) executes at a lower clock frequency than the mature ASICs, the time penalty caused by the cache-miss is further magnified in the modern SoCs.

[1] The workloads can not be cross-compiled using RISC-V toolchain.

[2] The counter does not interfere with the processor under test.

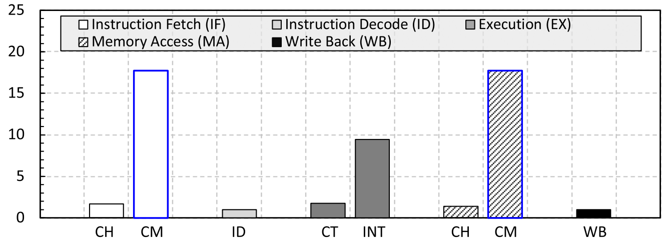


Fig. 3. Consumed clock cycles (averaged) at each stage (y-axis: cycles; CH: cache-hit; CM: cache-miss; CT: control operation; INT: integer computation).

B. L1 Cache Size and Task Execution Time

To further understand how L1 cache affects a task's (τ_i) execution time, we then developed a measurement-based method to determine the relationships between L1 cache size (denoted by A_j, where $1 \leq j \leq A$ and A denotes the total number of cache ways) and the task's corresponding WCET (denoted by $C_i[A_j]$).[3] The A_j and $C_i[A_j]$ are defined:

- *L1 cache size (A_j):* When task τ_i executes, it individually accesses A_j cache ways, i.e., $A_j \times a$ capacity of the L1 cache, a denotes the size of a cache way (unit: KB).
- *Task WCET with A_j ($C_i[A_j]$):* With A_j cache ways, task τ_i's corresponding WCET becomes $C_i[A_j]$.

The experiment-based method (illustrated in Fig. 4(a)) contains three steps:

Step 1: Initialization and input generation: We configured the size of D-cache to be A_j. We then initialized the processor using only the examined task τ_i and randomly generated N (e.g., 1,000) input data for the task.

Step 2: Experimental measures: We executed τ_i with the generated input data and recorded the task execution time for each run. We then compared all recorded results determining the maximum value to be $C_i[A_j]$.

Step 3: Iteration and plotting. We repeated the above steps with tuned A_j. Lastly, we plotted a diagram to show $C_i[A_j]$ under different A_j.

Example of a CNN Task: We used a CNN task to demonstrate the measurement-based method; the task was built on LeNet-5 architecture and trained using MNIST training dataset [33]. The task was quantized with full integer computations and measured on our experimental platform (Xilinx VC707, using the configurations given in Section VI). Also, we configure the size of a cache way to 2 KB (i.e., $a = 2$). Fig. 4(b) shows the experimental results, plotting the relationships between A_j and $C_i[A_j]$. As shown, the size of the L1 cache could significantly vary the task's WCET. In the example, the variance reached nearly 70%.

C. Research Challenge

Given the previously detailed concepts, it is possible to build a new real-time system, featuring a shared and re-sizable L1 cache, which facilitates the dynamic adjustment of task execution times via allocation of cache capacity. These new features offer the opportunity to unlock the potential parallelism and improve the throughput of the system. However, building such a real-time system is not straightforward, necessitating a comprehensive hardware-software full-stack approach. The research challenges can be summarized as follows:

- At the hardware level, a novel *micro-architecture* of the L1 cache is required to enable cache sharing and re-sizing among different processors. This new micro-architecture should avoid introducing any critical paths in the system, e.g., those that could affect the maximum system frequency or increase cache latency. At the same time, it must support timely capacity allocation to enable software-level allocation.
- At the software level, new *schedulability analysis* frameworks are required to theoretically evaluate the schedulability yielded by the new hardware. With that, a *selection algorithm* is needed to allocate cache capacity to the processors to simultaneously guarantee global schedulability and maximize system-wide throughput.
- As a systematic solution, a complete *system architecture* is necessary to build the real system, realizing the above features. In terms of real-world deployments, such an architecture must strike a balanced trade-off among schedulability, performance, overhead, and scalability.

In response to these challenges, we introduce a novel system framework (*Hopscotch*) for highly parallel real-time systems, incorporating a shared and re-sizable L1 cache (*Hopscotch-Cache*), along with a cache configuration algorithm. We will discuss them respectively below.

III. *Hopscotch*: Overview

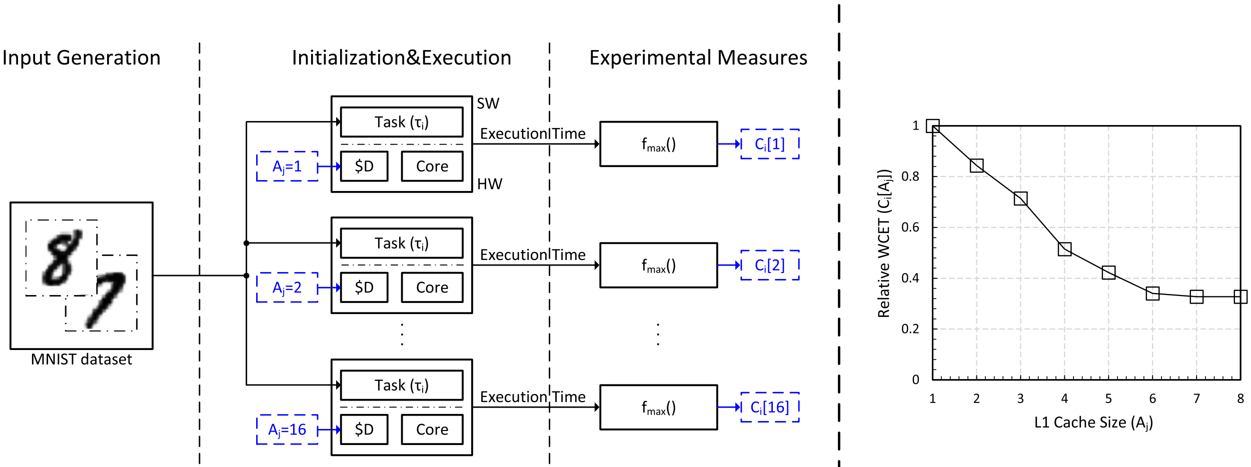
This section gives an overview of *Hopscotch*, presenting the top-level design concepts, system architecture and run-time behaviors of *Hopscotch*.

Context: In this paper, we make the following assumptions: i) the platform is an embedded Network-on-Chip (NoC); although *Hopscotch* is agnostic to the type of interconnect, deployment of a NoC can enhance the predictability of on-chip transactions [39]; ii) As an example, the paper presents the design and analysis for run-time re-sizable D-cache. Although the presented methods are also suitable for I-Cache and L2 cache.

A. Design Concepts

To take advantage of the observations given in Section II, we designed a new L1 cache (*Hopscotch-Cache*), shared between the processors in the same computing cluster (see Section IV), which enables run-time cache re-sizing across the processors. Based on *Hopscotch-Cache*, we established a new real-time system framework (see Section III-B), i.e., *Hopscotch*. *Hopscotch* dynamically allocates the cache capacity to tasks during context switches, unblocking the available parallelism. As a systematic solution, we further present a new theoretical model and schedulability analysis (see Section V), guaranteeing the system-level real-time schedulability.

[3] This method can be used for both I-cache and D-cache. In this paper, we use D-cache for demonstration purposes.



(a) Measurement-based method

(b) Example: a CNN task

Fig. 4. Determining the relationships between A_j and $C_i[A_j]$ (in Fig. 4(b), the y-axis is normalized by the processor without cache).

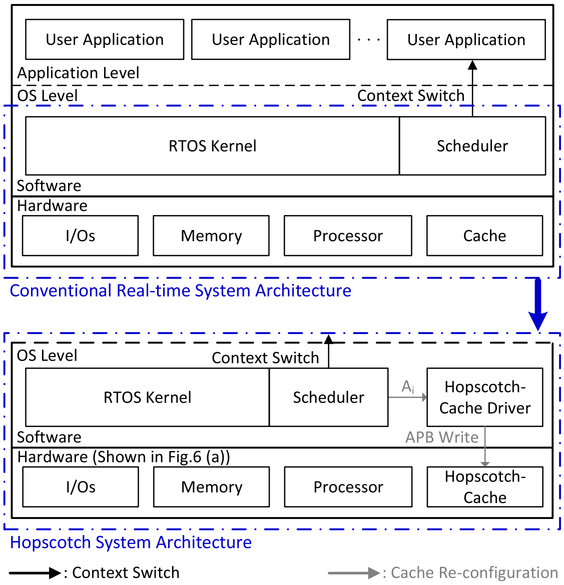


Fig. 5. Comparative system architectures.

B. System Architecture

As described in the design concepts, *Hopscotch* changes the system's architecture in both the hardware and software layers (Fig. 5), compared to a conventional real-time system.

Hardware Layer: In the hardware layer, we group processors as multiple computing clusters (see Fig. 6(a)), where each cluster contains four processors and one *Hopscotch-Cache*. The *Hopscotch-Cache* provides an independent communication interface for each processor, avoiding inter-processor interference while the cache is accessed. At the same time, we connect the clusters and memory using an open-source real-time NoC [39], allowing memory accesses when a processor encounters a cache-miss.

Software Layer: Corresponding to the cache design, we present a new *Hopscotch-Cache* driver (see Algorithm 2) at the Operating System (OS) level, providing configuration interfaces to the *Hopscotch-Cache* (In this work, we use FreeRTOS, but the specific choice of RTOS is not limited). Additionally, we slightly modified the implementation of the scheduler in the OS kernel, letting the scheduler alter the cache size (A_i) for the next executing task. The method for determining each task's A_i is given in Section V.

Compatibility: Although *Hopscotch* introduces a new system architecture, the design minimizes modifications to the software (see Fig. 5). Moreover, the design maintains the original OS-application interfaces presented by a conventional real-time system, thereby ensuring *source compatibility* and allowing tasks designed for a conventional real-time system to be directly migrated.

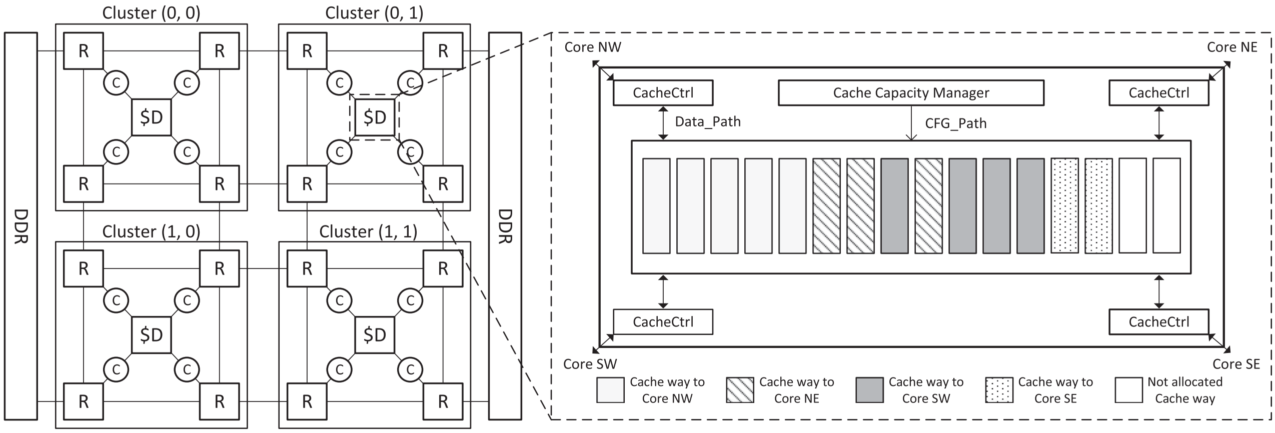
C. Run-Time Behaviors

At system initialization, the cache size required by each task (i.e., A_i) is pre-loaded (Algorithm 1: line 2). During context switches, the current task's context is stored, and the task with the highest priority is found from the ready queue as the next executing task (Algorithm 1: lines 7–10). The cache size demanded by the next task is then compared with the current task. If the next task requires a different cache size, the value of cache size is sent to *Hopscotch-Cache* using the *Hopscotch-Cache* driver (Algorithm 1: lines 11–17). Lastly, the next task's context is (re-)stored, and the task is executed by jumping the Program Counter (PC) to the specified address.

In *Hopscotch*, acquiring the run-time cache re-sizing relies on the *Hopscotch-Cache*; we therefore present the *Hopscotch-Cache* design details in the next section.

IV. *Hopscotch-Cache*: Hardware Design

In this section, we first recall the concepts of the set-associative cache architecture used by *Hopscotch-Cache*, then introduce the novel design of *Hopscotch-Cache*.



(a) NoC-based multi-core system (b) Top-level micro-architecture of Hopscotch-cache

Fig. 6. Hardware architecture of *Hopscotch* (C: Processor core; R: Router/Arbiter; $D: *Hopscotch-Cache*).

Algorithm 1: Context Switch in *Hopscotch*. Text in Blue Shows Additions in *Hopscotch*.

```
▷ Loading tasks' demanded cache size.
u8 A [NUM_TASKS];
▷ OS Kernel: context switch.
Function Context_Switch(task *current):
    task *next = NULL;
    Kernel.Intr.disable();
    ▷ Handling current task.
    Kernel.Context.save (current);
    ▷ Handling next task.
    next = Kernel.Find_next_task();
    if (A[next→ID] != A[current→ID]) then
        ▷ Acquiring the processor's ID.
        u8 hart_id;
        asm volatile ("csrr %0, mhartid" : "=r"(hart_id));
        ▷ Configuring cache size.
        Cache.Cfg_size (A[next→ID], hart_id);
    end
    Kernel.Context.store(next);
    current = next;
    Kernel.Intr.enable();
    Kernel.Context.jump_to_PC (current);
End Function
```

Set-Associative Architecture: In modern computing architectures, memory banks are organized as multiple blocks, with each block usually storing 32 or 64 bytes of data [25]. At execution, a memory block can be placed in a cache line using different schemes. The most popular placement scheme is *set-associative* [38], where a *set* is a group of cache lines, and a memory block can be placed anywhere within the mapped cache set (see Fig. 7). The cache architecture realizing this scheme with A cache lines in a set is called A-way set-associative architecture. Specifically, if the cache architecture only has one cache way, it is termed a direct-mapped architecture, as a memory block is always placed in the same location; if the cache architecture only has one set, it is termed a fully-associative architecture, as a block can be placed anywhere [25].

While mapping a memory block to a cache line, the block's address is divided into tag, index, and offset fields. The index field selects the cache set, the tag field finds the cache line, and the offset field determines the specific location of the data in the cache line (see Fig. 7).

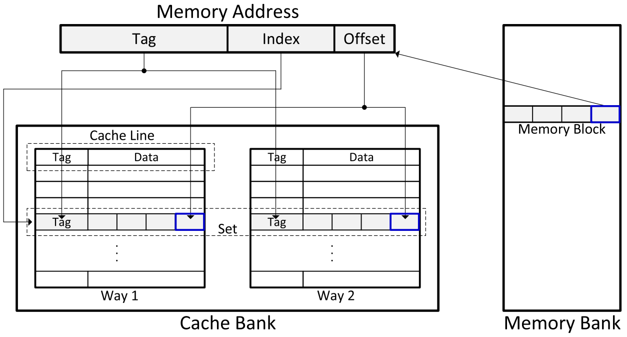
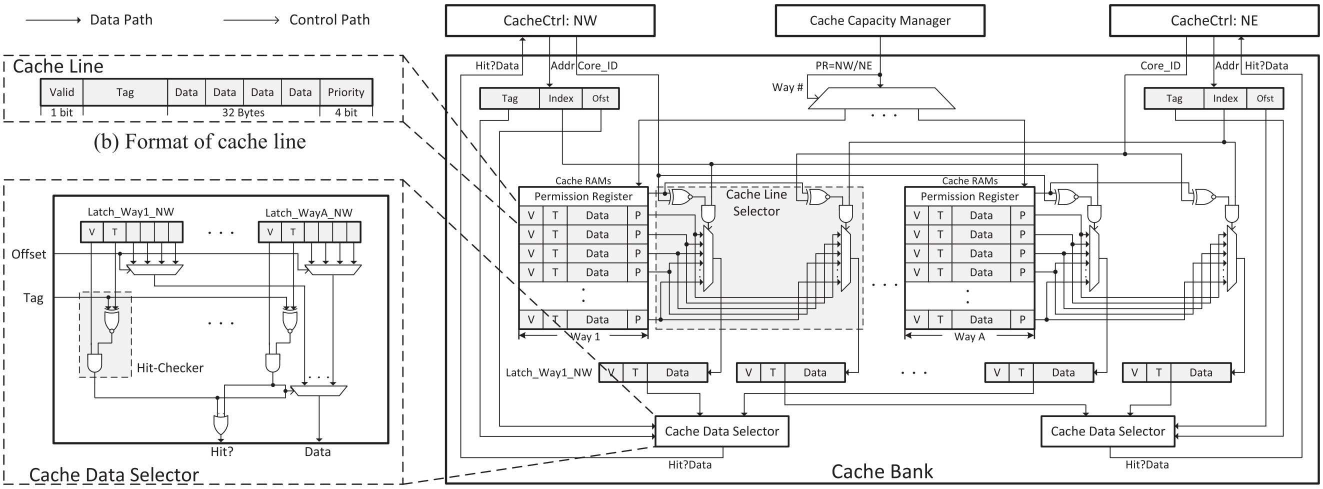


Fig. 7. A 2-way set-associative cache ($A = 2$).

A. Methods of Cache Re-Sizing

With set-associative architecture, it is possible to re-size a processor/task's cache capacity by reconfiguring the cache sets or cache ways it could use. Compared to reconfiguring ownership of the cache ways, reconfiguring ownership of the cache sets has two key issues: i) the width of the index field varies at run-time, as the index field locates the cache set; ii) extra identification and comparison circuits are necessary for each cache set to determine the set's ownership. These two issues significantly increase the design complexity, potentially leading the cache to become the system's critical path and decreasing the system's performance [45]. Hence, we designed *Hopscotch-Cache* with a reconfiguration of the cache ways. Specifically, we introduce a permission register for each cache way, recording the way's ownership. Processors can only access cache ways which they own. While configuring a processor/task's cache size, *Hopscotch-Cache* assigns more or fewer cache ways by modifying the values of the permission registers. We detail the *Hopscotch-Cache* design below.



(b) Format of cache line

(c) Micro-architecture of cache line selector

(a) Micro-architecture of cache bank

Fig. 8. Design of *Hopscotch-Cache* cache bank (V: Validness; T: Tag; P: Priority).

B. Hopscotch-Cache *Overview*

The typical use of *Hopscotch-Cache* in a NoC-based multi-core system is illustrated in Fig. 6(a), where *Hopscotch-Cache* is physically connected to the processors in the same computing cluster. From *Hopscotch-Cache*'s view, the processors are locally indexed using their relative locations, which are North-West (NW), North-East (NE), South-West (SW), and South-East (SE). At execution, *Hopscotch-Cache* manages the cache capacity and the cache accesses for these processors. To this end, we designed *Hopscotch-Cache* using three main modules (Fig. 6(b)):

- *Cache controller* – provides write and read interfaces between the processors and the cache bank.
- *Cache bank* – buffers the memory blocks recently accessed by the processors and handles cache accesses.
- *Cache capacity manager* – configures the cache size of each processor by managing the cache ways' ownership.

Since the cache controller does not require modification, we instantiate the standard cache controllers in *Hopscotch-Cache* and assign the controllers the same local IDs as the connected processors. Below, we present the design details of the cache bank and the cache capacity manager.

C. Design of Hopscotch-Cache *Bank*

The design of the *Hopscotch-Cache* bank (see Fig. 8(a)) mainly comprises cache RAMs, Cache Line Selectors (CLSs), Cache Data Selectors (CDSs) and cache replacement units.

Cache RAMs: We implement the Cache RAMs using Block RAMs provided by the experimental platform (Xilinx VC707) and organize the BRAMs using the set-associative architecture , where each cache set has A cache lines (i.e., A-way). Each cache line has four portions: valid, tag, data, and priority portions (Fig. 8(b)). The valid portion indicates the cache line's validness; the tag and data portions hold the tag and the data of the mapped memory block; the priority portion reveals the cache line's priority used in cache replacement. In addition, we introduce a permission register for each cache way, recording the cache way's ownership.

Cache Line Selector (CLS): We designed the CLS so that each individual CLS was associated with one cache way, checking whether the cache request has the corresponding permission. If the request is permitted, the CLS selects the cache line and forwards the selected cache line to the CDS. Otherwise, the CLS masks the request, preventing the request to access this cache way. To achieve this, we deploy groups of XNOR gates and AND gates, connecting the cache controllers and the cache way. Specifically, we connect the cache controller's ID bits with the permission register using the XNOR gate, and connect the XNOR gate's output with the cache controller's request path using the AND gate. Such connections mask requests issued by a cache controller without owning the way. In addition, we connect the cache lines using a multiplexer and connect the multiplexer's control port to the AND gate's output, which selects the cache lines using the request's index field. Since we designed the CLS using pure combinational logic, it consistently completes the filtering and selection in a single clock cycle.

Cache Data Selector (CDS): We designed the CDS so that each individual CDS was associated with one cache controller (Fig. 8(c)), checking whether the issued request meets a cache-hit. If there is a cache-hit, the CDS returns the corresponding data and a cache-hit signal. Otherwise, the CDS returns a cache-miss signal. To this end, we deploy A latches to buffer the CLS outputs and connect a hit-checker to each latch, which comprises an XNOR gate and an AND gate. Specifically, we connect the latch's tag portion and the request's tag field using the XNOR gate, checking the status of the cache-hit, and connect the XNOR gate's output and the latch's valid portion using the AND gate, checking the validness of the cache line (buffered in the latch). In addition, we connect the latches' data portions using multiplexers. The multiplexers select the latches' data portion using the request's offset field when the hit-checker feeds a hit signal. Like

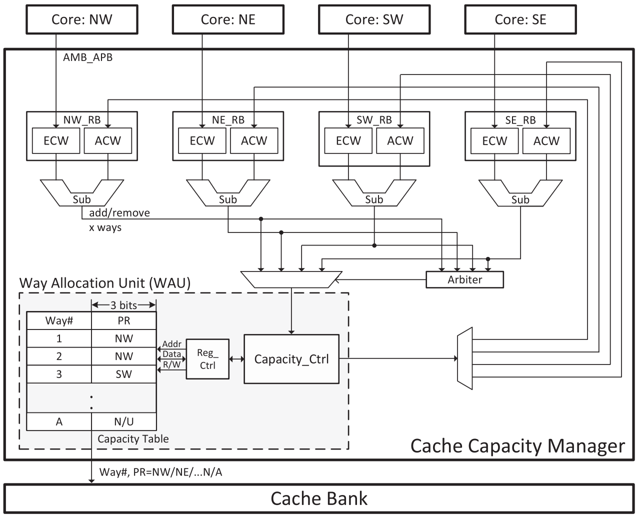


Fig. 9. Design of cache manager (RB: Register Bank; Ctrl: Controller).

the CLS, we designed the CDS using pure combinational logic, ensuring the cache-hit checking and data selecting to complete in a single clock cycle.

Cache Replacement Units: Hopscotch-Cache bank is compliant with different replacement policies. Since we reserved a priority portion in each cache line, the cache replacement units can always replace the cache line with the lowest priority. The only difference while deploying these replacement policies is the priority assignment. For example, with the Least Recently Used (LRU) policy, cache lines are prioritized using the reverse order of accesses. Following the methods described in [38], we implement three cache replacement units for *Hopscotch-Cache*, using LRU, RSU, FIFO, and Not Most Recently Used (NMRU) policies, respectively. We also evaluate *Hopscotch-Cache*'s real-time performance with these replacement units in Section VI.

D. *Design of* Hopscotch-Cache *Manager*

The design of *Hopscotch-Cache* manager is shown in Fig. 9, mainly comprising groups of register banks, a Way Allocation Unit (WAU), and an arbiter.

Register Banks: We designed the register bank so that each individual register bank was associated with one processor. A register bank has two registers, storing the processor's Expected Cache Ways (ECW register) and Actual Cache Ways (ACW register). We connect the ECW register to an AMBA APB interface and map it to a dedicated memory address [5]. This allows the software to modify the ECW register directly using memory write operations. Algorithm 2 shows the software driver for configuring the ECW register. Also, we connect a subtractor to each register bank, calculating the gap between the ECW and ACW registers. If the result does not equal zero, the gap value is sent to the WAU.

Way Allocation Unit (WAU): The WAU comprises a capacity table and a capacity controller in Fig. 9. The capacity table is the "shadow" of the permission registers in the cache bank, and the capacity controller changes the cache ways' ownership by writing to the capacity table. When adding cache ways, the capacity controller only writes to the free slots in the table, which are labeled "N/U"; when reducing the cache ways, the capacity controller writes "N/U" to the corresponding slot(s), which are now free. The slots are selected in a round-robin manner. Once the capacity table is updated, the new value is directly mapped to the corresponding permission register, and the capacity controller also updates the ACW register simultaneously. Note that when the ACW register's value is not equal to the WAU register's value, the processor still executes tasks with an unexpected cache size. In Section VII-C, we specifically evaluate this configuration latency.

Algorithm 2: Driver for Cache Size Configurations.

```
1 Function Cfg_size($u8$ $size$, $u32$ $core\_id$):
2     $u32$ $core\_id\_cluster$, $cluster\_id$;
3     Global2Local($core\_id$,
4             $\&core\_id\_cluster$, $\&cluster\_id$);
5     $u32$ $offset$ = 0x1000 $\times$ $\&cluster\_id$ |
6             0x04 $\times$ $id\_in\_cluster$;
7     $u32$ $addr\_ewc\_reg$ = BASEADDR_CACHE | $offset$;
8     Mem.wr($addr\_ewc\_reg$, $size$);
9 End Function
```

Arbiter: Since the register banks (i.e., ECW and ACW registers) may generate the gap values at the same time, we designed an arbiter to schedule the pending gap values. The arbiter's execution follows two rules: i) for gap values with different signs, the negative gap values are always served first; ii) for gap values with the same sign, the gap values are served in a Round-Robin manner. Rule (i) ensures cache ways can be sufficiently used by the processors requiring them, and rule (ii) ensures the processors' capacity requests can be fairly served.

Software Interfaces: Algorithm 2 illustrates the software driver for *Hopscotch-Cache*, providing a uniform interface accessible by all processor cores. Given that the majority of control complexity is managed by the hardware, the software driver's implementation is relatively straightforward. It only requires the processor IDs and their anticipated cache sizes as the input parameters (Line 1). These processor IDs are utilized to compute the offsets for their corresponding ECW registers (Lines 2–6) and to establish their mapped addresses (Line 7). Once these calculations are complete, the processor's requested cache capacity can be conveyed to *Hopscotch-Cache* via memory write operations (Line 8).

So far we have described *Hopscotch*'s system architecture and design methods, to ensure the real-time schedulability of *Hopscotch*, we now present the theoretical model and schedulability analysis in the next section.

V. Schedulability Analysis and Cache Size Selection

In the system of interest to be analyzed, we use M to denote the total number of processors and use A to denote the total number of cache ways, where M and A are given constant integers for a given system. On such a platform, we consider the scheduling of n cache-aware sporadic real-time tasks, each of which is modelled as follows.

Task Model: We model a cache-aware sporadic real-time task τ_i by a 4-tuple (A_i, C_i, D_i, T_i). Each task τ_i releases a (potentially infinite) sequence of *jobs* with a minimum separation of T_i time units between any consecutive jobs and T_i is called the *period* of τ_i. Each job of τ_i requires to occupy one processor and A_i cache ways to commence its execution, has a worst-case execution time (WCET) of C_i time units (while using one processor and A_i cache ways), and has an (absolute) deadline at D_i time units after its release time. C_i and D_i are also called the WCET and the relative deadline of task τ_i. We denote an arbitrary job of task τ_i by J_i, which is released at time r_i and has an absolute deadline at $d_i = r_i + D_i$. A job is called *waiting* if it is released but not executing. In this paper, we focus on constrained-deadline tasks only, where it is assumed that $\forall i, D_i \leq T_i$. In the schedulability analysis presented in this section, the 4-tuple of every task in the system is considered as given constants – A_i and C_i are obtained from experiment-based methods mentioned in Section II-B, and D_i and T_i are given as task specification.

Scheduling Rules: We focus on the non-preemptive global earliest-deadline-first (NP-GEDF) scheduling, where ready jobs are sorted in the *waiting queue* by their absolute deadlines from earlier ones to later ones and deadline ties are broken arbitrarily. At every event (job release or job completion), the scheduler checks the jobs in the waiting queue one by one in order. At the time when a job J_i is being checked by the scheduler, if there is at least one processor and at least A_i cache ways available, the scheduler immediately dispatches J_i to commence execution, J_i is removed from the waiting queue, and the number of available processors and cache ways are reduced accordingly. Once dispatched, a job will execute non-preemptively until completion.

Non-Blocking Waiting Queue: Note that, according to the scheduling rules above, for two jobs J_i and J_j such that $d_i < d_j$, it is possible that J_j is being dispatched to execute while J_i remains in the waiting queue. This may happen if $A_i > A_j$ and at the time of dispatching the number of available cache ways is less than A_i but at least A_j. Therefore, we say that our waiting queue is a *non-blocking* one. — The front waiting job does not block other jobs (with lower priorities) from being dispatched to execute. By contrast, if a *blocking* waiting queue is adopted, no waiting job can be dispatched to execute when the job at the front of the waiting queue is waiting due to insufficient available cache ways.

Parameter Δ_i: For each task τ_i, we define a parameter Δ_i to denote the *maximum* number of available caches ways when a job of τ_i is prevented from commencing execution due to insufficient available cache ways (i.e., assuming a processor is available to τ_i already). It is clear that $(A_i - 1)$ is a safe upper-bound on Δ_i. Nonetheless, for given constant A and constants $\{A_i\}$ in a system being analyzed, Δ_i can be derived more precisely. [15] has provided a dynamic programming algorithm[4] for calculating such precise Δ_i with a time complexity of $\mathcal{O}(A^2 \cdot n)$ where A is the total number of cache ways and n is the number of tasks. Note that, for a given system, $\{\Delta_i\}$ can be obtained offline in a prior to the schedulability analysis and therefore in the rest of this section, $\{\Delta_i\}$ are also treated as task-attribute constants.

In the rest of this section, we provide a linear programming (LP) based schedulability test. The analysis framework is inspired by [21] with the following major differences.

- [21] focused on fixed-priority scheduling while we investigate EDF scheduling;
- [21] adopted a blocking waiting queue setting while we consider a non-blocking waiting queue;
- [21] did not introduce and leverage the $\{\Delta_i\}$ parameters.

Job of Interest: To analyze the schedulability, we restrict our focus to an arbitrary job J_k (of task τ_k). Our goal is to derive a sufficient condition that ensures J_k meets its deadline. Without loss of generality,[5] we assume that

(P) All deadlines earlier than J_k's are met.

Problem Window: To investigate the execution of J_k, we focus on the time interval $[r_k, s_k]$, where $s_k = r_k + D_k - C_k$ is the latest time instant J_k must start execution in order to meet its deadline. The time interval $[r_k, s_k]$ is called our *problem window* and its length is clearly $(D_k - C_k)$. Because of non-preemptive scheduling, if J_k starts its execution at any point within the problem window, it will execute continuously until completion and meet its deadline. Furthermore, at any time point in the schedule of the problem window, if (i) a processor is available and (ii) at least A_k cache ways are available, J_k would have been scheduled to start execution at that point because a non-blocking waiting queue is adopted. Therefore, we further consider sub-intervals in the problem window in the following two categories.

- *α-interval*, where all the M processors are occupied;
- *β-interval*, where less than M processors are occupied but available cache ways are not sufficient for J_k.

Note that, according to the definitions above, no α-interval would overlap with a β-interval. Therefore, letting $\ell_\alpha (\ell_\beta$, respectively) denote the accumulative length of all α-intervals (β-intervals, respectively) in the problem window, $\ell_\alpha + \ell_\beta = D_k - C_k$, which is the length of the problem window, is necessary for J_k to miss its deadline. Thus, $\ell_\alpha + \ell_\beta < D_k - C_k$ is a sufficient schedulability condition for J_k to meet its deadline. To this end, we use an LP to find an upper bound on $\ell_\alpha + \ell_\beta$, subject to constraints to be presented next.

For each task τ_i other than τ_k, we introduce two variables α_i and β_i, where α_i (β_i, respectively) denotes the accumulative execution time of task τ_i in α-intervals (β-intervals, respectively) in the problem window. For task τ_i, we count its jobs possibly being executed in the problem window in three categories.

- *Carry-in job* that has release time before but deadline within the problem window. Because constrained deadlines and *(P)* are assumed, at most one such job may execute in the problem window.

[4] [15] does not consider cache-aware tasks but address gang tasks that need to simultaneously occupy multiple processors to commence execution. Nonetheless, the idea and notion of Δ_i can be seamlessly adapted to concern cache ways, and the algorithm for calculating Δ_i directly applies.

[5] By induction on jobs in deadline order, a schedulability test assuming *(P)* is sufficient to guarantee all deadlines are met.

Algorithm 3: Heuristic for Selecting $\{A_i\}$.

```
for $i = 1$ to $n$ do
    $A_i = 1$;
    for $a = 2$ to $A$ do
        if $(C_i[a-1] - C_i[a])/T_i \geq \theta$ then
            $A_i = a$;
        end
    end
end
```

- *Body job* that has both release time and deadline within the problem window. There are at most $\lfloor (D_k - C_k)/T_i \rfloor$ jobs executing in the problem window.
- *Carry-out job* that has release time within but deadline after the problem window. There is at most one such job executing in the problem window.

Therefore, the accumulative execution time of task τ_i in the problem window is upper bounded by $(\lfloor (D_k - C_k)/T_i \rfloor + 2) \cdot C_i$, and we have the first set of LP constraints as follows.

$$\forall i : i \neq k :: \alpha_i + \beta_i \leq \left(\left\lfloor \frac{D_k - C_k}{T_i} \right\rfloor + 2 \right) \cdot C_i \tag{1}$$

To further identify more constraints, we define *cumulative processor area (CPA)* and *cumulative cache area (CCA)* for a time interval in a schedule as follows. Letting $\mathtt{np}(t)$ denote the number of occupied processors at time t, the CPA of the time interval $[t_1, t_2]$ is defined by $\int_{t_1}^{t_2} \mathtt{np}(t)dt$. Letting $\mathtt{nc}(t)$ denote the number of occupied cache ways at time t, the CCA of the time interval $[t_1, t_2]$ is defined by $\int_{t_1}^{t_2} \mathtt{nc}(t)dt$.

The summation of CPAs of all α-intervals in the problem window can be calculated by $\sum_{i \neq k} \alpha_i$ by the definition of α_i and the fact that any task occupies exact one processor when executing. On the other hand, by the definition of α-interval and ℓ_α, The summation of CPAs of all α-intervals in the problem window can also be calculated by $M \cdot \ell_\alpha$. Therefore, we have the following constraint.

$$\sum_{i \neq k} \alpha_i = M \cdot \ell_\alpha \tag{2}$$

The summation of CCAs of all β-intervals in the problem window can be calculated by $\sum_{i \neq k} (A_i \cdot \beta_i)$ by the definition of β_i and the fact that task τ_i occupies A_i cache ways when executing. On the other hand, in a β-interval, J_k is prevented from executing only due to insufficient available cache way. Therefore, by the definition of Δ_k, at most Δ_k ($\Delta_k \leq A_k - 1$) cache ways are available, that is, at least $(A - \Delta_k)$ cache ways are occupied, at any time instant in a β-interval. In this direction of calculation, The summation of CCAs of all β-intervals in the problem window is at least $(A - \Delta_k) \cdot \ell_\beta$. Thus, we have the following constraint.

$$\sum_{i \neq k} (A_i \cdot \beta_i) \geq (A - \Delta_k) \cdot \ell_\beta \tag{3}$$

TABLE I
Hardware Experimental Setup

Core	Single-width, Speculative, 5-stage pipeline, RISC-V Freedom E31 [3]
Pipeline	1Int Alu, 1 FP/Div/Mult Alu, 1 MEM, 1 Jump unit
Branch Pred.	TAGE algorithm, 28-entry BTB, 512-entry BHT, 6-entry RAS
	Memory
L1 I$	4KB/core, 4-way, 1∼2 cycles
L1 D$	32KB/cluster, 8/16-way, 1∼2 cycles
LLC	4MB, 16-way, 15∼25 cycles
Memory	4GB, DDR3, max 32 requests, 40∼70 cycles
	Interconnects
NoC	BlueShell [39], full-duplex, max 32 requests

Lastly, recall the definitions of α_i, β_i, ℓ_α, ℓ_β, and notice the fact that the accumulative execution time of a task in the α-intervals (β-intervals, respectively) in the problem window cannot exceed the accumulative length of all α-intervals (β-intervals, respectively) in the problem window. The last two constraint sets follow.

$$\forall i : i \neq k :: \alpha_i \leq \ell_\alpha \tag{4}$$

$$\forall i : i \neq k :: \beta_i \leq \ell_\beta \tag{5}$$

Summary: For each task τ_k, we construct the following LP:

$$\begin{aligned} &\text{maximize} && \ell_\alpha + \ell_\beta \\ &\text{subject to} && (1)\text{–}(5) \end{aligned}$$

where $A_i, C_i, D_i, T_i, \Delta_i$ for all i as well as M and A are given constants, and α_i, β_i for all $i \neq k$ plus ℓ_α, ℓ_β are a total of $2(n-1) + 2 = n$ non-negative variables. Also, there are a total of $(n-1) + 1 + 1 + (n-1) + (n-1) = (3n-1)$ linear constraints in constraints sets (1)–(5). By constructing and solving this LP for every task in the system, we can conclude a sufficient schedulability test, which is presented as the following theorem.

Theorem 1: For each task τ_k, we solve an LP as constructed above and let χ_k denote the value of the optimization solution. The cache-aware sporadic task system is schedulable if

$$\forall k, \chi_k < D_k - C_k.$$

Selecting $\{A_i\}$: As noted earlier, the above schedulability test is applicable for any given set of $\{A_i\}$. We now briefly discuss how $\{A_i\}$ could be selected. Leveraging the experiment-based methods (discussed in Section II-B), we have profiled the WCET of each task τ_i for any selection of A_i, we denote the WCET of τ_i when $A_i = a$ $(1 \leq a \leq A)$ as $C_i[a]$. One way to obtain the optimal selection of $\{A_i\}$ is to iterate all the A^n combinations of $\{A_i\}$ and to apply the schedulability analysis in this section for every combination. Note that, for offline analysis, exponential time complexity might not be excessively forbidden. Nonetheless, in case such exponential time complexity is unacceptable (e.g., the number of tasks n is large), we instead apply a heuristic to select $\{A_i\}$ as described in Algorithm 3, where θ is a tunable threshold parameter of system designer's choice and the time complexity is $\mathcal{O}(n \cdot A)$. The intuition behind Algorithm 3 is that giving more cache ways to a task may reduce its execution time and therefore reduce its utilization to benefit the system schedulability; however, this also means that this task occupying more cache ways may have a higher chance to block other tasks from execution, which could jeopardize the system schedulability. Therefore, this is a tradeoff where we need to decide whether it is worth allocating an additional cache way to this task. We use θ to quantify this tradeoff and each θ value is a heuristic to select a set of $\{A_i\}$. Note that we have conducted extensive experiments in Section VII-A to evaluate the impact on the system's schedulability under different values of θ.

VI. Evaluation: Overhead and Scalability

In this section, we conduct experiments to examine *Hopscotch*'s overhead and scalability.

Experimental Platform: The hardware experimental setup is summarized Table I. We built 8/16-core *Hopscotch* variants on a Xilinx VC707 evaluation board. *Hopscotch*$|kway$-X denotes *Hopscotch* with k-way *Hopscotch-Cache* (data cache) and X cache replacement policy. We implemented the processors based on SiFive Freedom E31 [3], an open-source 32-bit RISC-V processor, and configured the processors to support 5-stage pipelined and in-order instruction processing. We also allocated an independent instruction cache to each processor with a fixed 4 KB capacity. We implemented the *Hopscotch-Cache* and related modules using Chisel [7], compiled into Verilog [47]. We connected the processors, *Hopscotch-Cache*, and external memory (4 GB DRAM) using a 5 × 5 mesh type open-source NoC [39], constructing the hardware using the topology illustrated in Fig. 6. The hardware was synthesized using Vivado (v.2021.1). We selected FreeRTOS (v.10.4) as the OS kernel for all processors, with the modifications introduced in Section III-B. The software (OS kernels, drivers, and user applications) was compiled using a RISC-V GNU tool-chain.

As described in Section III, the real-time performance of existing multi-core systems relies on the task scheduling presented at the OS level. Therefore, we built two Baseline Systems (BS)s on similar hardware platforms using conventional cache design, allocating each processor independent data cache. Each data cache had a fixed cache capacity which was $1/4$ of the *Hopscotch-Cache* presented in *Hopscotch* variants and instantiated with LRU replacement policy. BS | OSK is a baseline system implementing task scheduling at the OS kernels, and BS | HYP is a baseline system using virtualization, including real-time patches, implementing task scheduling in a dedicated hypervisor [53]. All systems ran at 100 MHz.

A. Software Overhead

In this section, we examine the software overhead of BS | OSK, BS | HYP, and *Hopscotch*.

Experimental Setup: The software overhead was evaluated using run-time memory footprint [24]. We first compared the memory footprint of the OS kernel used by the examined systems, where the kernels were fully-featured. Since *Hopscotch* needs an additional driver for *Hopscotch-Cache*, we compared the driver memory footprint of *Hopscotch-Cache* with other commonly used drivers, including drivers of Video Memory Direct Access (VDMA) and HDMI (1.4) controllers, examining the overhead of the *Hopscotch-Cache* driver from the system perspective. The tool used in experiments was a RISC-V GNU tool-chain.

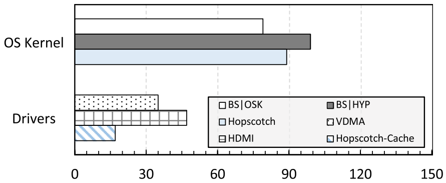


Fig. 10. Software overhead (unit: KB), which is evaluated via memory footprint, containing segments of BSS, data and text.

Obs. 1: The kernel in *Hopscotch* used more software overhead than BS | OSK's kernel, and slightly less than BS | HYP's kernel. *Hopscotch-Cache* driver's overhead was negligible.

This observation is shown in Fig. 10. The OS kernel in *Hopscotch* consumed an additional 10 KB (12.7%) memory footprint compared to the BS | OSK's OS kernel. This is because *Hopscotch*'s kernel is built on the BS | OSK's kernel, but involves additional cache management in the scheduler (detailed in Section III-B). Compared to the BS | HYP's kernel, *Hopscotch*'s kernel consumed slightly less overhead, as BS | HYP requires additional implementation to support the hypervisor. Although *Hopscotch* required the extra cache driver, its software overhead is lower than the other drivers. As shown in Fig. 10, the *Hopscotch-Cache* driver only consumed 17.4 KB memory footprint, which is 63.8% and 51.4% lower than the VDMA driver and HDMI drivers.

B. FPGA Overhead

As *Hopscotch* required extra hardware implementation for the *Hopscotch-Cache*. We evaluate its hardware overhead.

Experimental Setup: We configured *Hopscotch-Cache* variants with 32 KB capacity and 16 cache ways. As a *Hopscotch-Cache* is shared between four processors, we compared *Hopscotch-Cache*'s overhead with four conventional cache modules used in BS | OSK and BS | HYP. The conventional cache module was instantiated from Freedom E310 SoC, containing a cache controller and a cache bank (8 KB, 4 ways). We also examined the *Hopscotch-Cache*'s overhead along with other hardware elements, including a RISC-V tile (excluding the cache module), and two mainstream I/O controllers (VDMA and HDMI), examining *Hopscotch-Cache*'s overhead from the system perspective. The I/O controllers were chosen from the standard Xilinx IP library (with default settings). All components were synthesized and implemented by Vivado (v2020.2) and compared for Look-Up-Tables (LUTs), registers, and BRAMs. Since these metrics were evaluated using different units, we normalized the experimental results using the summation of four conventional cache modules: 2,131 LUTs, 847 registers, and 12 BRAMs.

Obs. 2: Hopscotch-Cache used more hardware overhead than the conventional cache modules. The extra overhead is considered acceptable compared to other hardware elements.

As shown in Fig. 11, the *Hopscotch-Cache* variants consumed an additional 30%–70% LUTs and 25%–35% registers,

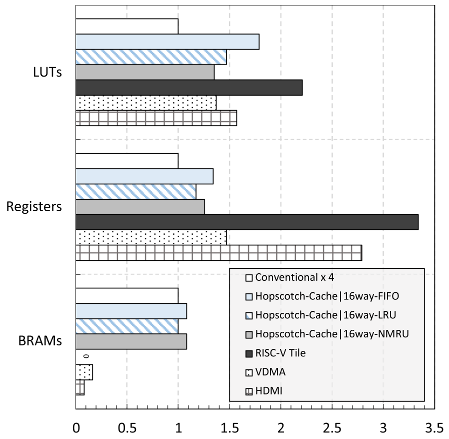


Fig. 11. Hardware overhead, normalized by 4× conventional cache modules.

compared to the conventional cache modules. Such overhead is mainly caused by deploying the additional logic (e.g., cache capacity manager) to support run-time reconfiguration. Among *Hopscotch-Cache* variants, *Hopscotch-Cache*$|16way$-*LRU* required the most hardware overhead, due to the complexity of implementing the LRU policy. From the system perspective, *Hopscotch-Cache* design is resource-efficient. *Hopscotch-Cache*$|16way$-*LRU*'s overhead was still less than other hardware elements: RISC-V tile (35.2% LUTs, 64.7% registers), VDMA (107.3% LUTs, 79.9% registers), HDMI (93.7% LUTs, 42.1% registers).

C. ASIC Overhead

To examine the overhead of *Hopscotch-Cache* in ASIC deployments, we conducted a physical implementation of a 16-core SoC (400 Mhz) using *Hopscotch-Cache* (32 KB, 16 ways per cluster) and conventional L1 cache (8 KB, 4 ways per processor). The physical implementation was carried out at the post-layout stage using Synopsys $28nm$ Generic PDKs [20]. The RTL was synthesized using the Synopsys Design Compiler (v2022.12), and the resulting netlist was placed and routed with Synopsys IC Compiler 2 (v2022.12).

Obs. 3: It is feasible to integrate *Hopscotch-Cache* into a 16-core SoC, which results in a slight increase in the SoC area.

The *Hopscotch* (i.e., the SoC) has a reported area of 2.701 mm^2, with each cluster accounting for 0.515 mm^2. Within a cluster, the four processors occupy 0.354 mm^2, while the *Hopscotch-Cache* occupies 0.078 mmÂ^2. In comparison, the SoC designed using the conventional L1 cache has a reduced total area of 2.591 mm^2, attributable to its simpler cache microarchitecture. In summary, developing a 16-core SoC with the *Hopscotch-Cache* results in an additional 0.11 mm^2 consumption, representing 4.24% of the SoC's area.

D. Scalability

Because the scalability impacts the feasibility of a proposed design, we examine the hardware scalability of Hopscotch using a varying number of processors.

Experimental Setup: We used the same method described in Section VI-B to implement the *Hopscotch* variants (i.e., the systems built upon *Hopscotch-Cache*) and a legacy system (i.e., a multi-core system using conventional cache modules) with a scaling number of processors. We chose the LRU replacement policy for the *Hopscotch* variants, as implementing the LRU consumed more hardware overhead than other replacement policies. Additionally, we introduced a scaling factor: η to control the number of processors (2^η). We first compared the scalability of area consumption between the legacy system, *Hopscotch* variants, and the correspondingly introduced cache design in *Hopscotch* variants. The area consumption was normalized by the overall area of the experimental platform, including LUTS, registers, and BRAMs. We then examined the scalability of power consumption, calculated as the sum of static and dynamic power simulated by the tool. Lastly, we evaluated the maximum frequency of the *Hopscotch-Cache* across the legacy system using varying η.

Obs. 3: The *Hopscotch-Cache*'s area and power consumption were linearly scaled by η. Compared to the legacy system, using *Hopscotch-Cache* slightly increased the area and power.

As seen in Fig. 12(a), when the experiments were scaled with η, the area consumption of *Hopscotch-Cache* was linearly scaled. This benefits from the resource-efficient design illustrated in Section IV. Although deploying the *Hopscotch-Cache* in *Hopscotch* used more hardware than the legacy system, the introduced area consumption was within 17%. Power consumption is affected by voltage, clock frequency, toggle rate and design area [25]. Since the unified voltage, clock frequency and simulated toggle rate were assigned by the tool, the design area dominated the elements' power consumption. As expected, power consumption increased linearly when η increased (see Fig. 12(b)).

Obs. 5: When scaled with η, deploying the *Hopscotch-Cache* did not affect the maximum frequency.

This observation is shown in Fig. 12(c): the maximum frequency of the *Hopscotch-Cache* variants decreased with increasing η, but was always higher than the legacy system. This indicates that the *Hopscotch-Cache* did not become a critical path, and did not reduce maximum system frequency.

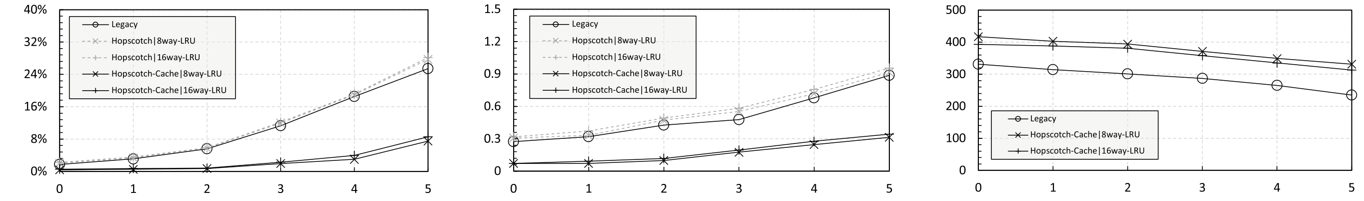
VII. Evaluation: Real-Time Performance

We now use real-world use cases to evaluate the real-time performance of the examined systems. The experiments were carried out on the same platform discussed in Section VI.

System Configurations: We configured the systems with 8/16 processors. For the *Hopscotch* variants, we configured each *Hopscotch-Cache* with 32 KB capacity and 8/16 ways. For the BS | OSK and BS | HYP, we configured each cache module with 8 KB capacity and 2/4 ways.

Task Sets: We deployed three sets of software tasks:

- 10 automotive safety tasks, selected from the Renesas automotive use case database [17], including CRC-32, RSA-32, and core-self test, etc.
- 10 automotive function tasks, selected from the EEMBC benchmark [16], including, Fast Fourier Transform (FFT), speed calculation, etc.



(a) Area consumption (y-axis: percentage). (b) Power consumption (y-axis: Watts (W)). (c) Maximum frequency (y-axis: MHz).

Fig. 12. Scalability: area, power, and maximum frequency versus scaling factor η.

- Synthetic workloads built on LeNet-5 architectures, and trained using MNIST, EMINST, and CIFAR-10 training datasets [33]. The synthetic workloads can be added to the system to control overall utilization.[6]

For baseline systems (BS | OSK and BS | HYP), we employed a hybrid-measurement approach to obtain the tasks' WCETs (C_is). Each task had a randomly defined period (R_i), with overall processor utilization of approximately 45%. All tasks were assigned using implicit deadlines. For *Hopscotch* variants, we used the method introduced in Section II-B to find the tasks' WCETs ($C_i[A_j]$s) under different cache sizes (A_js), then adopted the heuristic presented in Section V to determine the most suitable A_i for each task.

Before the experiments, the raw data processed by the tasks was randomly generated and stored in the external memory. During the experiments, the processors fetched the raw data and sent the calculated results back to the external memory. For a fair comparison, we ensured the data input to the examined systems was identical in each execution.

A. Cache Size Selection

We observed how tunable threshold (θ), introduced by the heuristic in Section II-B, affected *Hopscotch*'s schedulability, then selected the A_is of the tasks for the following experiments.

Experimental Setup: We first determined the tasks' A_is using Algorithm 3 with different θs, where $\theta \in [0, 0.7]$ (at intervals of 0.05). We then executed the task sets and synthetic workloads on the *Hopscotch* variants 100 times, with 70% target utilization (the mean value used in the following experiments). We evaluated the examined systems using the *success ratio*, recording the percentage of trials that executed successfully (i.e., without deadline misses of any safety or function tasks) under a specified target utilization.

Obs. 6: When $\theta \in [0.3, 0.35]$, the selected A_is ensured *Hopscotch*-variants achieved the best real-time performance.

As shown in Fig. 14, the tunable threshold (θ) significantly varied the schedulability of *Hopscotch*. Such variance reached nearly 80%. When θ was equal to 0.3 or 0.35, the found A_is ensured *Hopscotch*-variants achieved the best real-time performance. Therefore, in the following experiments, we configured θ to be 0.3 or 0.35 to determine the tasks' A_is and $C_i[A_i]$s. As shown in Fig. 14, we configured θ to be different values, observing how θ affected the success ratio. We then used the θ for each *Hopscotch* variant which led to the best result, so that each access variant had its own θ.

B. Real-Time Performance

Experimental Setup: We introduced two groups of experimental setups, activating 8/16 processors to execute the task sets and synthetic workloads. In each experimental group, we executed each examined system 100 times under varying target utilization [45%–95% at intervals of 5%. We evaluated the examined systems using *success ratio* under a specified target utilization. Each run lasted 300 seconds.

Obs. 7: Hopscotch variants outperformed the baseline systems using the same experimental settings.

As shown in Fig. 13(a) and (b), when the systems were configured with the same settings (i.e., core number and target utilization), *Hopscotch* variants continuously achieved higher success ratios than the baseline systems (BS | OSK and BS | HYP). Such improvements benefited from deploying the *Hopscotch-Cache* (described in Section IV) and allocating suitable cache sizes to the tasks, unblocking the available parallelism and improving the system-level real-time schedulability.

Obs. 8: In *Hopscotch* variants, adjusting the number of cache ways has more impact on the systems' real-time performance than adjusting the replacement policies.

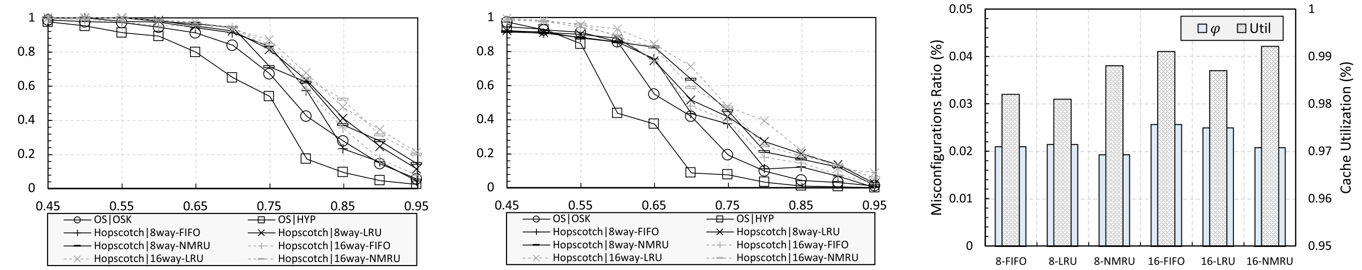
As shown in Fig. 13(a) and (b), *Hopscotch*$|16way$ usually acquired 5%-10% higher success ratios than the *Hopscotch*$|8way$ under the same settings. For the replacement policies, we observed that *Hopscotch* variants using the FIFO policy had the worst real-time performance of all the test cases, at 3% lower than the *Hopscotch* variants using LRU and NMRU on average.

C. Analysis of Side Effects

The development of the *Hopscotch-Cache* fundamentally modifies the features of the L1 cache, enabling both cache sharing and resizing. Although previous experiments have demonstrated the schedulability improvements brought by these new features, they may still affect the effectiveness of L1 cache, especially during busy system periods. Such impacts are summarized in two main domains: i) a reduction in L1 cache utilization due to the need for additional cache management; and ii) delays in cache resizing caused by frequent contentions between processors. Therefore, we present an effectiveness analysis to examine these impacts.

Experimental Setup: We adopted the same experimental setup and methods introduced in Section VII-B, with only

[6] Notably, since the task's practical execution time can be affected by diverse factors, adding synthetic workloads only gives the system a *target utilization*.



(a) Real-time performance (8-core systems).
(b) Real-time performance (16-core systems).
(c) Analysis of Side Effects (16-core systems).

Fig. 13. Case study. In (a) and (b), x-axis: target utilization; y-axis: the success ratio. In (c), y-axis: average φ.

Hopscotch variants being executed. To replicate a high-demand scenario, we configured *Hopscotch* with 16 cores and 100% utilization. In addition, we deployed a cycle-accurate monitor to trace the processors and *Hopscotch-Cache* in each computing cluster, recording (i) the utilization of *Hopscotch-Cache* and (ii) the latency of resizing. We evaluated utilization by calculating the percentage of the cache ways that have been occupied. Resizing latency, on the other hand, was assessed by determining the percentage of task executions that occurred with an unexpected cache capacity, represented as φ. For example, if task τ_i was executed in 9 ms with A_i and 1 ms with another cache capacity (due to resizing latency), φ_i is 10%. Note that we classified false positive cases as the correct configuration, as it accelerated the associated task's execution.[7]

Obs. 9: When systems were in busy period, the *Hopscotch-Cache* could be fully utilized.

The observation was given by Fig. 13(c), revealing that when the systems were configured for 100% utilization, the average cache utilization across all scenarios exceeded 98%. These findings verify the effectiveness of *Hopscotch-Cache* which safeguards the systems' throughput.

Obs. 10: The average resizing latency affected about 2% of task executions; adding more cache ways increased such latency.

As shown in Fig. 13(c), with all experimental settings, the φ was always less than 2.7%. However, when *Hopscotch* was configured with more cache ways, the φ also increased slightly, indicating a higher resizing latency. This is mainly caused by the cache capacity manager (which can only configure one cache way at one-time point), requiring more clock cycles to set the cache ways correctly.

D. Summary

In the current and previous sections (Section VI), we have examined *Hopscotch* in terms of real-time performance, overhead, and scalability. The experimental results reveal that the introduction of a shared and re-sizable L1 cache, along with the capacity allocation algorithm, can improve system-wide schedulability across all test scenarios. Different cache replacement policies have minor impacts on the real-time performance of the *Hopscotch*, with variances remaining less than 3%. The implementation of *Hopscotch-Cache* based on the conventional set-associative cache incurs approximately 50% additional combinational logic and 30% sequential logic, as well as 20 K software overhead (kernel modifications and software drivers). The partial sharing of *Hopscotch-Cache* effectively ensures its hardware scalability, which does not impact the system's critical path.

VIII. Related Work

A. L1 Cache Sharing

L1 cache sharing has been considered in both academia and industry. In academia, Nakajima et al. [36] introduced a shared L1 cache for dual-core SoCs. As one of the early attempts, this work verified the feasibility of implementing the shared L1 cache. Rahimi et al. [42] presented a logarithmic interconnect to connect processors and multi-banked L1 cache, successfully enabling L1 cache sharing between 32 processors. However, the interconnect brought multiple critical paths into the system, causing significant time penalties while accessing the cache. Kakoee et al. [29] extended the work of Rahimi et al. by integrating controllable pipelines between the processors and the cache banks, effectively breaking the critical paths and reducing the time penalties. Kakoee et al. [30] then further updated the design of the cache controller, reducing cache access latency down to one clock cycle. However, as reported by [30], the method also brought an additional 40% hardware and energy overhead. Considering energy-efficiency, Gautschi et al. [18] and Witting et al. [52] upgraded the cache design and the scheduling methods for cache accesses, mitigating the extra energy consumption caused by L1 cache sharing. Overall, the existing research on shared L1 cache mainly focuses on scalability, throughput, and energy-efficiency. However, L1 cache resizing and the system's real-time schedulability have not been studied.

In industry, shared L1 cache has also been deployed in many commercial SoCs, e.g., STM's STHORM MPSoC [8] and Plurality's HyperCore [48]. The most successful examples are Maxwell [1] and Pascal [2], GPU families designed by NVIDIA. These architectures feature many Streaming Multiprocessors (SMs), where an SM contains 128 or 64 CUDA processors. The CUDA processors in the same SM share the same L1 cache. In commercial SoCs, the shared L1 cache is usually adopted to enhance the system's throughput. However, as in the academic work, L1 cache resizing and system-level schedulability are not considered.

[7] False positive: a task executes with more cache capacity than A_i.

Furthermore, although we have focused on sequential tasks only in this paper in order to investigate both the design details and analysis in depth, we would like to point out that the ideas of sharing and managing L1 cache have the potential to benefit parallel tasks even more significantly because the parallel threads of the same task may share even more data. Due to space and scope limits, we leave further investigation on parallel tasks to future work.

B. L1 Cache Resizing

There has been work exploring L1 cache resizing, but this is mainly focused on energy-efficiency and security. In terms of energy efficiency, Cai et al. [11] demonstrated how L1 cache resizing affects a system's performance and energy. Following this work, Wang et al. [49], [50] proposed a scheduling-aware L1 cache resizing method, gating certain cache banks based on the workloads. This method reduced the L1 cache's energy consumption up to 74%. In terms of security, Huang et al. [26], [27] partitioned the L1 cache into protected and unprotected regions for different types of tasks, and dynamically adjusted the sizes of these regions to balance the system's vulnerability and energy-efficiency. However, none of the above work considered L1 cache sharing and system-level schedulability.

C. L2 Cache and Last Level Cache (LLC) Partitioning

Different from the L1 cache, L2 cache and Last Level Cache (LLC) are originally designed to be shared between processors. Hence, there is more work that studied cache partitioning for the L2 cache and LLC, which could be briefly classified into two groups: i) throughput-aware partitioning, ii) energy-aware partitioning, and iii) schedulability-aware partitioning.

Throughput-Aware Partitioning: Throughput-aware cache partitioning is a vital research direction that aims at improving system-wide performance by finding the appropriate allocation of the L2 cache or LLC. For example, Qureshi et al. [41] presented a utility-based cache-partitioning scheme to increase LLC's average throughput. The scheme regulated that each software application enforces the creation of a new LLC partition, and each partition in the system is dynamically reshaped according to the utility curves. Based on [41], Jaleel et al. [28] and Xie et al. [54], [55] presented different cache replacement policies to further reduce the contentions between the running applications. To improve the granularity of the cache partitioning, Manikantan et al. [34] and Sanchez et al. [44] introduced fine-grained schemes to partition the cache for each thread. Similar to this work, many other schemes were presented to allocate L2 cache or LLC to threads and assign threads' priority correspondingly, e.g., [10], [12], [22], [37]. However, none of the work studied system-level schedulability.

Energy-Aware Partitioning: In terms of energy-efficiency, both static and dynamic schemes were presented to partition L2 cache and LLC. Specifically, Reddy et al. [43] profiled software applications offline and determined their cache requirements. With that, a method was presented to determine cache partitioning, optimizing global energy-efficiency. However, with the ever-increase software complexity, static schemes become unrealistic [46]. For dynamic partitioning, Albonesi et al. [4], and Sundararajan et al. [45], [46] proposed different cache designs that can vary its size and associativity by enabling or disabling cache ways or sets. Powell et al. [40] utilized the voltage gating technology to disable unused cache lines to reduce the dynamic power. Following this work, Meng et al. [35] further studied the scheme's impacts on power leakage. Similarly, Ghosh et al. [19] and Kedzierski et al. [32] presented different schemes to partition the LLC to improve both static and dynamic power consumption globally. However, real-time schedulability was not considered.

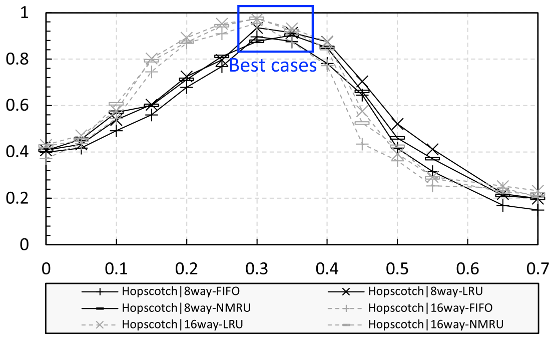


Fig. 14. Success ratio under different θs (x-axis: θ; y-axis: success ratio).

Schedulability-Aware Partitioning: With the consideration of the system-level schedulability, Kim et al. [31] presented a method to allocate L2 cache and LLC across the processors, trying to optimize the cache partitioning and system-level schedulability simultaneously. Guo et al. [23] presented a mixed Integer Linear Program (ILP) with approximation algorithms to partition shared cache and then mapped applications with strong cache interference onto different processors. Unlike this work, researchers also studied application-level partitioning for L2 cache and LLC. For example, Guan et al. [21] presented a method using cache coloring to partition the LLC to each application with non-preemptive global scheduling. This work was also extended with preemptive scheduling [56]. Similarly, Chen et al. [14] allocated L2 cache to specific tasks and presented an ILP to create a time-triggered scheduling method, minimizing the cache misses for a pre-allocated taskset. In contrast to the introduced work that utilized existing technology (e.g., cache coloring) to partition L2 cache, this paper presents a systematic solution for L1 cache, including design, analysis, and configuration, achieving more fine-grained trade-off between the cache size, WCET, and system-level schedulability (e.g., refer to Section II, Figs. 4 and 14). Additionally, our *Hopscotch-Cache* design allows the run-time resizing, providing flexibility to respond to the run-time system changes, e.g., tasks join or leave the system.

D. RISC-V Processors

RISC-V is an Instruction Set Architecture (ISA) that developed from the University of California, Berkeley [51], marking a significant shift in ISA design paradigms. Unlike mainstream ISAs, e.g., Intel's x86 or ARM, a salient feature of RISC-V is its modularity. This allows for a tailored approach, letting designers incorporate only the ISA components pertinent to their needs. This modular design caters to a wide range of applications, from compact embedded systems to powerful supercomputers. For example, the *Hopscotch-Cache* configurations (discussed in Section IV-D) also leverage such modularity for further acceleration. Consistent with RISC principles, RISC-V also adopts a minimalistic core instruction set, emphasizing a lean yet versatile set of instructions, often resulting in more efficient hardware implementations. Since RISC-V's inception, numerous microarchitectures have been presented, including Rocket [6], BROOM [13], SonicBoom [57], and Freedom E31 [3]. It's also noteworthy that the proposed *Hopscotch-Cache* is not exclusive to RISC-V processors; it is also flexible to be implemented across other architectures.

IX. Conclusion

In this paper, a novel re-sizable L1 cache is presented, enabling partial cache sharing and run-time cache re-sizing between processors. With the L1 cache design, a novel system framework (*Hopscotch*) is proposed for highly-parallel multi-core systems. *Hopscotch* dynamically allocates L1 cache capacity to the tasks executed on the processors, unblocking the available parallelism and ensuring system-level real-time schedulability. Corresponding to the system framework, a new theoretical model and schedulability analysis are presented to provide a timing guarantee for *Hopscotch*. As shown in the evaluations, *Hopscotch* effectively improves system-level schedulability compared to conventional real-time systems. In addition, *Hopscotch* is resource-efficient.

References

[1] NVDIA Maxwell Architecture, 2022. Accessed: Apr. 12, 2022. [Online]. Available: https://developer.nvidia.com/maxwell-compute-architecture

[2] NVDIA Pascal Architecture, 2022. Accessed: Apr. 12, 2022. [Online]. Available: https://www.nvidia.com/en-gb/data-center/pascal-gpu-architecture/

[3] SiFive RISC-V, 2022. Accessed: Apr. 12, 2022. [Online]. Available: https://github.com/sifive/soc-freedom-sifive

[4] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," in *Proc. ACM/IEEE 32nd Annu. Int. Symp. Microarchitecture*, 1999, pp. 248–259.

[5] "AMBA AXI and ACE protocol specification," 2012. [Online]. Available: https://documentation-service.arm.com/static/602a9df190ee6824a1e02b98?token=

[6] K. Asanovic et al., "The rocket chip generator," EECS Dept., Univ. California, Berkeley, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2016–17, 2016.

[7] J. Bachrach et al., "Chisel: Constructing hardware in a Scala embedded language," in *Proc. 49th Annu. Des. Automat. Conf.*, 2012, pp. 1212–1221.

[8] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Proc. Des. Automat. Test Europe Conf. Exhib.*, 2012, pp. 983–987.

[9] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The PARSEC benchmark suite: Characterization and architectural implications," in *Proc. 17th Int. Conf. Parallel Architectures Compilation Techn.*, 2008, pp. 72–81.

[10] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *Proc. IEEE/ACM 41st Int. Symp. Microarchitecture*, 2008, pp. 318–329.

[11] Y. Cai, M. T. Schmitz, A. Ejlali, B. M. Al-Hashimi, and S. M. Reddy, "Cache size selection for performance, energy and reliability of time-constrained systems," in *Proc. Asia South Pacific Des. Automat. Conf.*, 2006, pp. 923–928.

[12] F. J. Cazorla, P. M. Knijnenburg, R. Sakellariou, E. Fernandez, A. Ramirez, and M. Valero, "Predictable performance in SMT processors: Synergy between the OS and SMTs," *IEEE Trans. Comput.*, vol. 55, no. 7, pp. 785–799, Jul. 2006.

[13] C. Celio, P.-F. Chiu, K. Asanović, B. Nikolić, and D. Patterson, "BROOM: An open-source out-of-order processor with resilient low-voltage operation in 28-nm CMOS," *IEEE Micro*, vol. 39, no. 2, pp. 52–60, Mar./Apr. 2019.

[14] G. Chen, B. Hu, K. Huang, A. Knoll, D. Liu, and T. Stefanov, "Automatic cache partitioning and time-triggered scheduling for real-time MPSoCs," in *Proc. Int. Conf. Reconfigurable Comput. FPGAs*, 2014, pp. 1–8.

[15] Z. Dong and C. Liu, "Analysis techniques for supporting hard real-time sporadic gang task systems," *Real-Time Syst.*, vol. 55, no. 3, pp. 641–666, 2019.

[16] EEMBC, "EEMBC benchmark," 2015. [Online]. Available: https://www.eembc.org/autobench/

[17] R. Electronics, "Renesas: Automotive use cases," Accessed: Apr. 12, 2022, 2022. [Online]. Available: https://www.renesas.com/eu/en/solutions/automotive/technology/safety.html

[18] M. Gautschi, D. Rossi, and L. Benini, "Customizing an open source processor to fit in an ultra-low power cluster with a shared L1 memory," in *Proc. 24th Ed. Great Lakes Symp. VLSI*, 2014, pp. 87–88.

[19] M. Ghosh, E. Ozer, S. Ford, S. Biles, and H.-H. S. Lee, "Way guard: A segmented counting bloom filter approach to reducing energy for set-associative caches," in *Proc. ACM/IEEE Int. Symp. Low Power Electron. Des.*, 2009, pp. 165–170.

[20] R. Goldman, K. Bartleson, T. Wood, K. Kranen, V. Melikyan, and E. Babayan, "32/28nm educational design kit: Capabilities, deployment and future," in *Proc. IEEE Asia Pacific Conf. Postgraduate Res. Microelectronics Electron.*, 2013, pp. 284–288.

[21] N. Guan, M. Stigge, W. Yi, and G. Yu, "Cache-aware scheduling and analysis for multicores," in *Proc. 7th ACM Int. Conf. Embedded Softw.*, 2009, pp. 245–254.

[22] F. Guo, Y. Solihin, L. Zhao, and R. Iyer, "A framework for providing quality of service in chip multi-processors," in *Proc. IEEE/ACM 40th Annu. Int. Symp. Microarchitecture*, 2007, pp. 343–355.

[23] Z. Guo, K. Yang, F. Yao, and A. Awad, "Inter-task cache interference aware partitioned real-time scheduling," in *Proc. 35th Annu. ACM Symp. Appl. Comput.*, 2020, pp. 218–226.

[24] P. B. Hansen, *Operating System Principles*. Englewood Cliffs, NJ, USA: Prentice-Hall, 1973.

[25] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*. Amsterdam, The Netherlands: Elsevier, 2011.

[26] Y. Huang and P. Mishra, "Vulnerability-aware energy optimization for reconfigurable caches in multitasking systems," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 38, no. 5, pp. 809–821, May 2019.

[27] Y. Huang and P. Mishra, "Vulnerability-aware dynamic reconfiguration of partially protected caches," in *Proc. 21st Int. Symp. Qual. Electron. Des.*, 2020, pp. 255–260.

[28] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," *ACM SIGARCH Comput. Architecture News*, vol. 38, no. 3, pp. 60–71, 2010.

[29] M. R. Kakoee, I. Loi, and L. Benini, "A resilient architecture for low latency communication in shared-L1 processor clusters," in *Proc. Des. Automat. Test Europe Conf. Exhib.*, 2012, pp. 887–892.

[30] M. R. Kakoee, V. Petrovic, and L. Benini, "A multi-banked shared-L1 cache architecture for tightly coupled processor clusters," in *Proc. Int. Symp. Syst. Chip*, 2012, pp. 1–5.

[31] H. Kim, A. Kandhalu, and R. Rajkumar, "A coordinated approach for practical OS-level cache management in multi-core real-time systems," in *Proc. 25th Euromicro Conf. Real-Time Syst.*, 2013, pp. 80–89.

[32] K. Kkdzierski, F. J. Cazorla, R. Gioiosa, A. Buyuktosunoglu, and M. Valero, "Power and performance aware reconfigurable cache for CMPs," in *Proc. 2nd Int. Forum Next-Gener. Multicore/Manycore Technol.*, 2010, pp. 1–12.

[33] Y. LeCun et al., "LeNet-5, convolutional neural networks," 2015. [Online]. Available: http://yann.lecun.com/exdb/lenet

[34] R. Manikantan, K. Rajan, and R. Govindarajan, "Probabilistic shared cache management (PriSM)," in *Proc. 39th Annu. Int. Symp. Comput. Architecture*, 2012, pp. 428–439.

[35] Y. Meng, T. Sherwood, and R. Kastner, "On the limits of leakage power reduction in caches," in *Proc. 11th Int. Symp. High-Perform. Comput. Architecture*, 2005, pp. 154–165.

[36] M. Nakajima, T. Yamamoto, M. Yamasaki, K. Kaneko, and T. Hosoki, "Homogenous dual-processor core with shared L1 cache for mobile multimedia SoC," in *Proc. IEEE Symp. VLSI Circuits*, 2007, pp. 216–217.

[37] K. J. Nesbit, J. Laudon, and J. E. Smith, "Virtual private caches," in *Proc. 34th Annu. Int. Symp. Comput. Architecture*, 2007, pp. 57–68.

[38] D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC: The Hardware Software Interface*, 5th ed. San Mateo, CA, USA: Morgan Kaufmann.

[39] G. Plumbridge, J. Whitham, and N. Audsley, "Blueshell: A platform for rapid prototyping of multiprocessor NoCs and accelerators," *ACM SIGARCH Comput. Architecture News*, vol. 41, no. 5, pp. 107–117, 2014.

[40] M. Powell, S.-H. Yang, B. Falsafi, K. Roy, and T. Vijaykumar, "Gated-Vdd: A circuit technique to reduce leakage in deep-submicron cache memories," in *Proc. Int. Symp. Low Power Electron. Des.*, 2000, pp. 90–95.

[41] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. IEEE/ACM 39th Annu. Int. Symp. Microarchitecture*, 2006, pp. 423–432.

[42] A. Rahimi, I. Loi, M. R. Kakoee, and L. Benini, "A fully-synthesizable single-cycle interconnection network for shared-L1 processor clusters," in *Proc. Des. Automat. Test Europe*, 2011, pp. 1–6.

[43] R. Reddy and P. Petrov, "Cache partitioning for energy-efficient and interference-free embedded multitasking," *ACM Trans. Embedded Comput. Syst.*, vol. 9, no. 3, pp. 1–35, 2010.

[44] D. Sanchez and C. Kozyrakis, "Scalable and efficient fine-grained cache partitioning with vantage," *IEEE Micro*, vol. 32, no. 3, pp. 26–37, May/Jun. 2012.

[45] K. T. Sundararajan, T. M. Jones, and N. Topham, "Smart cache: A self adaptive cache architecture for energy efficiency," in *Proc. Int. Conf. Embedded Comput. Syst. Architectures Model. Simul.*, 2011, pp. 41–50.

[46] K. T. Sundararajan, V. Porpodas, T. M. Jones, N. P. Topham, and B. Franke, "Cooperative partitioning: Energy-efficient cache partitioning for high-performance CMPs," in *Proc. IEEE Int. Symp. High-Perform. Comput. Architecture*, 2012, pp. 1–12.

[47] D. Thomas and P. Moorby, *The Verilog Hardware Description Language*. Berlin, Germany: Springer, 2008.

[48] J. Turley, "Plurality, gets ambitious with 256 CPUs," Microprocessor Report, Jun. 2010.

[49] W. Wang and P. Mishra, "System-wide leakage-aware energy minimization using dynamic voltage scaling and cache reconfiguration in multitasking systems," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 20, no. 5, pp. 902–910, May 2012.

[50] W. Wang, P. Mishra, and A. Gordon-Ross, "Dynamic cache reconfiguration for soft real-time systems," *ACM Trans. Embedded Comput. Syst.*, vol. 11, no. 2, pp. 1–31, 2012.

[51] A. Waterman, L. Yunsup, P. A. David, and A. Krste, "The RISC-V instruction set manual, volume I: User-level ISA, version 2.0.,' EECS Dept., Univ. California, Berkeley, Tech. Rep. UCB/EECS-2014-54, 2014.

[52] R. Wittig, M. Hasler, E. Matus, and G. Fettweis, "Statistical access interval prediction for tightly coupled memory systems," in *Proc. IEEE Symp. Low-Power High-Speed Chips*, 2019, pp. 1–3.

[53] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: Towards real-time hypervisor scheduling in Xen," in *Proc. 9th ACM Int. Conf. Embedded Softw.*, 2011, pp. 39–48.

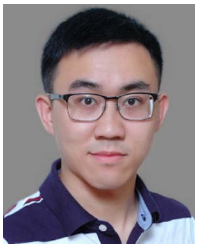
[54] Y. Xie and G. H. Loh, "PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches," *ACM SIGARCH Comput. Architecture News*, vol. 37, no. 3, pp. 174–183, 2009.

[55] Y. Xie and G. H. Loh, "Scalable shared-cache management by containing thrashing workloads," in *Proc. Int. Conf. High-Perform. Embedded Architectures Compilers*, Springer, 2010, pp. 262–276.

[56] M. Xu, L. T. X. Phan, H.-Y. Choi, and I. Lee, "Analysis and implementation of global preemptive fixed-priority scheduling with dynamic cache allocation," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, 2016, pp. 1–12.

[57] J. Zhao, B. Korpan, A. Gonzalez, and K. Asanovic, "SonicBOOM: The 3rd generation Berkeley out-of-order machine," in *Proc. 4th Workshop Comput. Architecture Res. RISC-V*, vol. 5, 2020, pp. 1–7.

Zhe Jiang (Member, IEEE) received the PhD degree from the University of York, in 2019. He is interested in the architecture, micro-architecture, and design automation for emerging computing systems, with a particular focus on improving functional safety, security, reliability, and timing-predictability of automotive, cloud and embedded computing systems.



Kecheng Yang received the BE degree in computer science and technology from Hunan University, in 2013, and the MS and PhD degrees from the University of North Carolina at Chapel Hill, in 2015 and 2018, respectively. He is an assistant professor with the Department of Computer Science, Texas State University. His research interests include real-time systems and scheduling algorithms. He received an Outstanding Paper Award and the Best Student Paper Award at the 40th IEEE RTSS, and an Oustanding Paper Award at the 26th RTNS.

Nathan Fisher received the BS degree from the University of Minnesota, in 1999, the MS degree from Columbia University, in 2002, and the PhD degree from the University of North Carolina at Chapel Hill, in 2007, all in computer science. He is an associate professor with the Department of Computer Science, Wayne State University. His research interests include real-time and embedded computer systems and approximation algorithms. He was the recipient of the NSF CAREER Award in 2010.

Nan Guan (Member, IEEE) received the PhD degree from Uppsala University, Sweden, in 2013. He is currently an Associate Professor with the Department of Computer Science, City University of Hong Kong. His research interests include real-time embedded systems and cyber-physical systems. He received ACM SIGBED Early Career Researcher Award in 2020, the EDAA Outstanding Dissertation Award in 2014, the Best/Oustanding Paper Award of RTSS 2009, DATE 2013, ACM e-Energy 2018, ISORC 2018, RTSS 2019, EMSOFT 2020, RTSS 2022.

Neil C. Audsley is currently deputy dean with the School of Mathematics, Computer Science and Engineering, City, University of London. His research interests include high performance real- time systems; real-time computer and memory architectures; real-time operating systems and their acceleration on FPGAs; timing analysis.

Zheng Dong received the BS degree from Wuhan University, China, in 2007, the MS degree from the University of Science and Technology of China, in 2011, and the PhD degree from the University of Texas at Dallas, USA, in 2019. He is an assistant professor with the Department of Computer Science, Wayne State University, Detroit, Michigan. His research interests include real-time embedded computer systems and connected autonomous driving systems. His current research focus is on multiprocessor scheduling theory and hardware-software co-design for real-time applications. He received the Outstanding Paper Award at the 38th IEEE RTSS. He is a member of the IEEE Computer Society.