Hopscotch: A Hardware-Software Co-design for Efficient Cache Resizing on Multi-core SoCs

Zhe Jiang, Kecheng Yang, Nathan Fisher, Nan Guan, Neil Audsley, 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.

1 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.

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].

- Zhe Jiang is with School of Integrated Circuits, Southeast University, China, 21000, and Computer Science Department, University of Cambridge, United Kingdom, CB3 0FD.
- Kecheng Yang is with the Department of Computer Science, Texas State University, San Marcos, TX 78666, United States.
- Nan Guan is with Computer Science Department, City University of Hong Kong, 83 Tat Chee Ave, Kowloon Tong, Hong Kong.
- Neil Audsley is with the Department of Computer Science, City, University of London, United Kingdom, EC1V 0HB.
- Zheng Dong and Nathan Fisher are with the Department of Computer Science, Wayne State University, Detroit, MI, 48202, United States.
§ Corresponding author, dong@wayne.edu.

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 τ1 and τ2 deployed in a dual-core system. At time point 0, τ1 and τ2 are released and executed by the processors simultaneously. At time point 2, τ1 completes the execution, and its L1 cache becomes free. However, due to the restriction, τ2 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 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-sizeable L1 cache (Fig. 1(b)). More cache capacity can be allocated to τ2 by “borrowing” the cache.
ways from $\tau_2$, accelerating $\tau_1$’s execution. Although this adjustment slows down $\tau_1$’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-sizeable 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: Sec. 2 presents the motivation and research challenges, followed by the design of Hopscotch in 3 and 4. Sec. 5 presents the theoretical analysis and cache size selection methods for Hopscotch. Sec. 6 and Sec. 7 evaluate the overhead and real-time performance of Hopscotch, respectively. Sec. 8 reviews the related work and Sec. 9 concludes the paper.

## 2 Motivation and Research Challenges

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

### 2.1 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]:

- **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).

### 2.2 L1 Cache Size and Task Execution Time

To further understand how L1 cache affects a task’s ($\tau_i$) 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 $C_i[A_j]$). The $A_j$ and $C_i[A_j]$ are defined:

1. The workloads can not be cross-compiled using RISC-V toolchain.
2. The counter does not interfere with the processor under test.
3. This method can be used for both I-cache and D-cache. In this paper, we use D-cache for demonstration purposes.
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.

3 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); 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.

3.1 Design Concepts

To take advantage of the observations given in Sec. 2, we designed a new L1 cache (Hopscotch-Cache), shared between the processors in the same computing cluster (see Sec. 4), which enables run-time cache re-sizing across the processors. Based on Hopscotch-Cache, we established a new real-time system framework (see Sec. 3.2), 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 Sec. 5), guaranteeing the system-level real-time schedulability.

3.2 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.

References

[33] Xilinx VC707, using the configurations given in Sec. 6.

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 $\tau_i$ and randomly generated $N$ (e.g., 1,000) input data for the task.

Step 2: Experimental measures. We executed $\tau_i$ 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 Sec. 6). 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%.

2.3 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.

Fig. 3. Consumed clock cycles (averaged) at each stage (y-axis: cycles; CH: cache-hit; CM: cache-miss; CT: control operation; INT: integer computation).
Fig. 4. Determining the relationships between $A_i$ and $C_i[A_j]$ (in Fig. 4(b), the $y$-axis is normalized by the processor without cache).

**Algorithm 1:** Context switch in Hopscotch. Text in blue shows additions in Hopscotch.

1. $\triangleright$ Loading tasks’ demanded cache size.
2. u8 A [NUM_TASKS];
3. $\triangleright$ OS Kernel: context switch.
4. Function Context_Switch(task *current):
   5. task *next = NULL;
   6. Kernel.Intr.disable();
   7. $\triangleright$ Handling current task.
   8. Kernel.Context.save (current);
   9. $\triangleright$ Handling next task.
   10. next = Kernel.Find_next_task();
   11. if (A[next→ID] != A[current→ID]) then
       12. $\triangleright$ Acquiring the processor’s ID.
       13. u8 hart_id;
       14. asm volatile ("csr %0, mhartid" : "r"(hart_id));
       15. $\triangleright$ Configuring cache size.
       16. Cache.Cfg_size (A[next→ID], hart_id);
   17. end
   18. Kernel.Context.store(next);
   19. current = next;
   20. Kernel.Intr.enable();
   21. Kernel.Context.jump_to_PC (current);
22. End Function

Software layer. Corresponding to the cache design, we present a new Hopscotch-Cache driver (see Algo. 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 Sec. 5.

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.

3.3 Run-time Behaviors

At system initialization, the cache size required by each task \((i.e., A_i)\) is pre-loaded (Algo. 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 (Algo. 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 (Algo. 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.

4 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.

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].
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.

### 4.2 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.

### 4.3 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 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 Sec. 6.

### 4.4 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. Algo. 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 Sec. 7.3, we specifically evaluate this configuration latency.

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. Algo. 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.

5 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 $\tau_i$ by a 4-tuple ($A_i, C_i, D_i, T_i$). Each task $\tau_i$ 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 $\tau_i$. Each job of $\tau_i$ 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 $\tau_i$. We denote an arbitrary job of task $\tau_i$ 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 – $A_i, C_i, D_i$ and $T_i$ are obtained from experiment-based methods mentioned in Sec. 2.2, 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 $\Delta_i$.** For each task $\tau_i$, we define a parameter $\Delta_i$ to denote the maximum number of available caches ways when a job of $\tau_i$ is prevented from commencing execution due to insufficient available cache ways (i.e., assuming a processor is available to $\tau_i$ already). It is clear that $(A_i-1)$ is a safe upper-bound on $\Delta_i$. Nonetheless, for given constant $A$ and constants $\{A_i\}$ in a system being analyzed, $\Delta_i$ can be derived more precisely. [15] has provided a dynamic programming algorithm$^4$ for calculating such precise $\Delta_i$ with a time complexity of $O(A^2 \cdot n)$ 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 $\tau_k$). 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 $[t_k, s_k]$, 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.

- **$\alpha$-interval,** where all the $M$ processors are occupied;
- **$\beta$-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 $\alpha$-interval would overlap with a $\beta$-interval. Therefore, letting

$4$. [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 $\Delta_i$ can be seamlessly adapted to concern cache ways, and the algorithm for calculating $\Delta_i$ directly applies.

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

<table>
<thead>
<tr>
<th>Algorithm 3: Heuristic for Selecting ${A_i}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. for $i = 1$ to $n$ do</td>
</tr>
<tr>
<td>2. $A_i = 1$;</td>
</tr>
<tr>
<td>3. for $a = 2$ to $A$ do</td>
</tr>
<tr>
<td>4. if $(C_i[a-1] - C_i[a]) / T_i \geq \theta$ then</td>
</tr>
<tr>
<td>5. $A_i = a$;</td>
</tr>
<tr>
<td>6. end</td>
</tr>
<tr>
<td>7. end</td>
</tr>
</tbody>
</table>

$\ell_\alpha$, $\ell_\beta$, respectively denote the accumulative length of all $\alpha$-intervals ($\beta$-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 $\tau_i$ other than $\tau_k$, we introduce two variables $\alpha_i$ and $\beta_i$, where $\alpha_i$ ($\beta_i$, respectively) denotes the accumulative execution time of task $\tau_i$ in $\alpha$-intervals ($\beta$-intervals, respectively) in the problem window. For task $\tau_i$, 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.
- **Body job** that has both release time and deadline within the problem window. There are at most $\lceil (D_k - C_k) / T_i \rceil$ 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 $\tau_i$ in the problem window is upper bounded by $(\lceil (D_k - C_k) / T_i \rceil + 2) \cdot C_i$, and we have the first set of LP constraints as follows.

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

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 $np(t)$ denote the number of occupied processors at time $t$, the CPA of the time interval $[t_1, t_2]$ is defined by \(\int_{t_1}^{t_2} np(t) dt\). Letting $nc(t)$ denote the number of occupied cache ways at time $t$, the CCA of the time interval $[t_1, t_2]$ is defined by \(\int_{t_1}^{t_2} nc(t) dt\).

The summation of CPAs of all $\alpha$-intervals in the problem window can be calculated by $\sum_{i \neq k} \alpha_i$ by the definition of $\alpha_i$ and the fact that any task occupies exactly one processor when executing. On the other hand, by the definition of $\alpha$-interval and $\ell_\alpha$, the summation of CPAs of all $\alpha$-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 all $\beta$-intervals in the problem window can be calculated by $\sum_{i \neq k} (A_i \cdot \beta_i)$ by the definition of $\beta_i$ and the fact that task $\tau_i$ occupies $A_i$ cache ways when executing. On the other hand, in a $\beta$-interval, $J_k$ is
are a total of $\Delta_k$, at most $\Delta_k$ (where $A_k \leq A_k - 1$) cache ways are available, that is, at least $(A - \Delta_k)$ cache ways are occupied, at any time instant in a $\beta$-interval. In this direction of calculation, the summation of CCAs of all $\beta$-intervals in the problem window is at least $(A - \Delta_k) \cdot \ell_\beta$. Thus, we have the following constraint.

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

(3)

Lastly, recall the definitions of $\alpha_i$, $\beta_i$, $\ell_\alpha$, $\ell_\beta$, and notice the fact that the accumulative execution time of a task in the $\alpha$-intervals ($\beta$-intervals, respectively) in the problem window cannot exceed the accumulative length of all $\alpha$-intervals ($\beta$-intervals, respectively) in the problem window. The last two constraint sets follow.

$$\forall i : i \neq k :: \alpha_i \leq \ell_\alpha$$

(4)

$$\forall i : i \neq k :: \beta_i \leq \ell_\beta$$

(5)

**Summary.** For each task $\tau_k$, we construct the following LP:

$$\begin{align*}
\text{maximize} & \quad \ell_\alpha + \ell_\beta \\
\text{subject to} & \quad (1) - (5)
\end{align*}$$

where $A_i, C_i, D_i, T_i, \Delta_i$ for all $i$ as well as $M$ and $A$ are given constants, and $\alpha_i, \beta_i$ for all $i \neq k$ plus $\ell_\alpha, \ell_\beta$ 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 $\tau_k$, we solve an LP as constructed above and let $\chi_k$ 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 Sec. 2.2), we have profiled the WCET of each task $\tau_i$ for any selection of $A_i$. We denote the WCET of $\tau_i$ 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 Algo. 3, where $\theta$ is a tunable threshold parameter of system designer’s choice and the time complexity is $O(n \cdot A)$. The intuition behind Algo. 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 $\theta$ to quantify this tradeoff and each $\theta$ value is a heuristic to select a set of $\{A_i\}$.

Note that we have conducted extensive experiments in Sec. 7.1 to evaluate the impact on the system’s schedulability under different values of $\theta$.

---

### 6 Evaluation: Overhead and Scalability

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

**Experimental Platform.** 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 4KB 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 (4GB DRAM) using a 5 x 5 mesh type open-source NoC [39], constructing the hardware using the topology illustrated in Fig. 6. The hardware was synthesized using Vivado (v2021.1). We selected FreeRTOS (v10.4) as the OS kernel for all processors, with the modifications introduced in Sec. 3.2. The software (OS kernels, drivers, and user applications) was compiled using a RISC-V GNU tool-chain.

As described in Sec. 3, 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) 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.

**6.1 Software Overhead**

In this section, we examine the software overhead of BS(OSK, BS|HYP, and Hopscotch.)
We configured 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 (8KB, 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. 1.** 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, 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'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).

**6.3 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 (32KB, 16 ways per cluster) and conventional L1 cache (8KB, 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 mm², with each cluster accounting for 0.515 mm². Within a cluster, the four processors occupy 0.354 mm², while the Hopscotch-Cache occupies 0.078 mm². In comparison, the SoC designed using the conventional L1 cache has a reduced total area of 2.591 mm², attributable to its simpler cache micro-architecture. In summary, developing a 16-core SoC with the Hopscotch-Cache results in an additional 0.11 mm² consumption, representing 4.24% of the SoC’s area.

**6.4 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 Sec. 6.2 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: $\eta$ to control the number of processors ($2^n$). 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 $\eta$.

Obs. 3. The Hopscotch-Cache's area and power consumption were linearly scaled by $\eta$. 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 $\eta$, the area consumption of Hopscotch-Cache was linearly scaled. This benefits from the resource-efficient design illustrated in Sec. 4. 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 $\eta$ increased (see Fig. 12(b)).

Obs. 5. When scaled with $\eta$, 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 $\eta$, 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.

7 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 Sec. 6.

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

For baseline systems (BS(OSK and BS(HYP), we employed a hybrid-measurement approach to obtain the tasks’ WCETs ($C_i$). 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 Sec. 2.2 to find the tasks’ WCETs ($C_i[A_j]$) under different cache sizes ($A_j$), then adopted the heuristic presented in Sec. 5 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.

7.1 Cache Size Selection

We observed how tunable threshold ($\theta$), introduced by the heuristic in Sec. 2.2, affected Hopscotch’s schedulability, then selected the $A_j$s of the tasks for the following experiments.

Experimental setup. We first determined the tasks’ $A_j$s using Algo. 3 with different $\theta$s, 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 $A_j$s ensured Hopscotch-variants achieved the best real-time performance.

As shown in Fig. 14, the tunable threshold ($\theta$) significantly varied the schedulability of Hopscotch. Such variance reached nearly 80%. When $\theta$ was equal to 0.3 or 0.35, the found $A_j$s ensured Hopscotch-variants achieved the best real-time performance. Therefore, in the following experiments, 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.
we observed that \( \phi \) usually acquired 5%-10% higher success ratios than the baseline systems. We then used the \( \theta \) for each \textit{Hopscotch} variant which led to the best result, so that each access variant had its own \( \theta \).

### 7.2 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.** \textit{Hopscotch} variants outperformed the baseline systems using the same experimental settings.

As shown in Figs. 13(a) and 13(b), when the systems were configured with the same settings (i.e., core number and target utilization), \textit{Hopscotch} variants continuously achieved higher success ratios than the baseline systems (BS|OSK and BS|HYP). Such improvements benefited from deploying the \textit{Hopscotch-Cache} (described in Sec. 4) and allocating suitable cache sizes to the tasks, unblocking the available parallelism and improving the system-level real-time schedulability.

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

### 7.3 Analysis of Side Effects.

The development of the \textit{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 Sec. 7.2, with only \textit{Hopscotch} variants being executed. To replicate a high-demand scenario, we configured \textit{Hopscotch} with 16 cores and 100% utilization. In addition, we deployed a cycle-accurate monitor to trace the processors and \textit{Hopscotch-Cache} in each computing cluster, recording (i) the utilization of \textit{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 \( \varphi \). For example, if task \( t_i \) was executed in 9 ms with \( A_i \) and 1 ms with another cache capacity (due to resizing latency), \( \varphi_i \) is 10%. Note that we classified false positive cases as the correct configuration, as it accelerated the associated task’s execution.\footnote{False positive: a task executes with more cache capacity than \( A_i \).}

**Obs. 9.** When systems were in busy period, the \textit{Hopscotch-Cache} could be fully utilized.

The observation was given by Figure 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 \textit{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 \( \varphi \) was always less than 2.7%. However, when \textit{Hopscotch} was configured with more cache ways, the \( \varphi \) 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.

7.4 Summary
In the current and previous sections (Sec. 6), 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.

8 RELATED WORK
8.1 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. 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. Kakoe et al. [29] extended the work of Rahimi et al. by integrating controlable pipelines between the processors and the cache banks, effectively breaking the critical paths and reducing the time penalties. Kakoe 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.

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.

8.2 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.

8.3 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 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. 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.

**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 Sec. 2, Fig. 4, and Fig. 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.

### 8.4 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 Sec. 4.4) also leverage such modularity for further acceleration. Consistent with RISC principles, RISC-V also adopts a minimalist 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.

---

### 9 Conclusion

In this paper, a novel re-sizeable L1 cache is presented, enabling partial cache sharing and run-time cache re-binning between processors. With the L1 cache design, a novel system framework (Hopscotch) is proposed for highly-parallel multicore 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.

---

### Acknowledgement

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, CNS-2104181.

### References


Zhe Jiang (Member, IEEE) received his Ph.D. from the University of York (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 in the Department of Computer Science at 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 Outstanding Paper Award at the 26th RTNS.

Nathan Fisher received the Ph.D. from the University of North Carolina at Chapel Hill in 2007, and M.S. degree from Columbia University in 2002, and the B.S. degree from the University of Minnesota in 1999, 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) is currently an associate professor at the Department of Computer Science, City University of Hong Kong. Dr Guan received his PhD from Uppsala University, Sweden in 2013. 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...

Zheng Dong received a 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 are in 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.

Neil C. Audsley is currently Deputy Dean of the School of Mathematics, Computer Science and Engineering at 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.