A Cache/Algorithm Co-design for Parallel Real-Time Systems with Data Dependency on Multi/Many-core System-on-Chips

Zhe Jiang[1], Shuai Zhao[2], Ran Wei[3,4], Yiyang Gao[2] and Jing Li[5]
[1]National Center of Technology Innovation for EDA, School of Integrated Circuits, South East University, China
[2]Sun Yat-Sen University, China [3]University of Cambridge, UK
[4]Lancaster University, UK [5]New Jersey Institute of Technology, US

Abstract

Parallel real-time systems rely on a shared cache for dependent data transmission. A conventional shared cache suffers from intensive interference, yet existing cache management techniques only ensure determinism for single-threaded tasks. This paper introduces a virtual indexed, physically tagged, selectively-inclusive, non-exclusive L1.5 Cache, offering way-level control and fine-grained sharing capabilities. Focusing on DAG tasks, we construct a scheduling method that exploits the L1.5 Cache to reduce data transmission, hence, the makespan. As a systematical solution, we built a real system, from the SoC and the ISA to the programming model. Experiments show that our solution significantly improves the timing performance of DAG tasks with negligible overheads.

1 Introduction

With ever-complex functionalities being implemented in real-time systems, multi/many-core architectures are increasingly applied, and tasks often have complicated execution dependencies. For instance, in an autonomous driving application, the control tasks must be executed after the completion of perception and decision tasks, forming a dependency graph due to the data flow [13, 14].

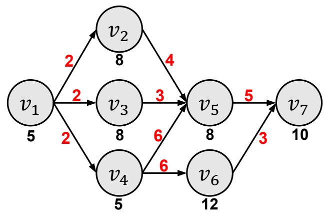


Figure 1: An example DAG task *(numbers in black: node computation time; numbers in red: edge communication cost).*

The Directed Acyclic Graph (DAG) model is one of the mainstream task models of parallel tasks with data dependency. In a DAG task, a node represents a series of computations that must be executed sequentially. An edge from node v_j to node v_k indicates the dependency between them — v_k can only start if v_j is finished and the produced data is transmitted to v_k with a communication cost associated with the edge [14]. Fig. 1 shows an example of a DAG task. When node v_1 is finished, it produces the data required for v_2, v_3 and v_4 to start execution, with a communication cost of 2.

Research on parallel real-time tasks focuses on the scheduling method based on their computation times [8, 15]. However, in

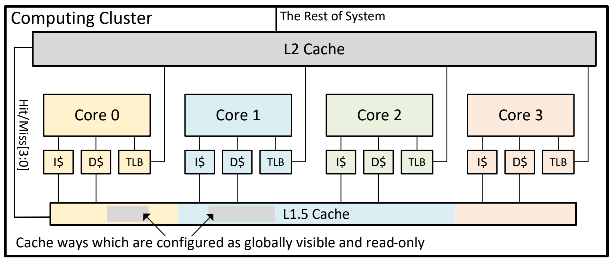


Figure 2: Deploying L1.5 Cache in a SoC, focusing only on the computing cluster. The colour-coding indicates the ownership of elements, with grey denoting shared ways. In this example, the L1.5 Cache is partitioned to be owned by cores 0 - 3, with certain ways set to be globally visible and read-only.

practice, the dependent data transmission heavily relies on the shared cache, which can incur significant communication cost due to inter-core cache interference, increasing the makespan. However, this is not well-recognised in the literature [5, 8, 15]. To address this issue, hardware support that facilitates data transmission in parallel tasks, whilst eliminating cache interference, is required.

Cache management techniques aimed at eliminating inter-core interference have been introduced, primarily to ensure the determinism of *single-threaded* tasks. Therefore, they either completely prohibit cache-level data sharing (e.g., colouring [4] or partitioning [7]) or just lock frequently-accessed data on shared cache statically (e.g., locking [12]). This imposes a significant barrier to supporting parallel tasks with complex data dependency, in which dependent data can be accessed from different cores.

Contributions. In this paper, (i) we present a *Virtual Indexed, Physically Tagged* (VIPT), *Selectively-Inclusive, Non-Exclusive* (SINE) L1.5 Cache, which we positioned in each computing cluster between the conventional L1 and L2 caches. The L1.5 Cache enables way-level reconfiguration of the ownership of the ways, global visibility and an inclusion policy, allowing managed and flexible cache sharing (Fig. 2). (ii) Focusing on the DAG tasks, we construct a novel scheduling method that leverages the L1.5 cache to reduce the communication cost and the DAG makespan. (iii) We built a systematic full-stack framework from the System-on-Chip (SoC) and Instruction Set Architecture (ISA) to the programming model, forming a complete solution for parallel real-time tasks.

We deployed our proposed system on the AMD Alveo U280 FPGA and examined it using various metrics, including makespan, success ratio, and overhead. Experiments show that deploying the L1.5 Cache in real-time SoCs can significantly improve the timing performance of DAG tasks with negligible hardware overhead.

2 L1.5 Cache: Overview

In this work, we make the following assumptions: (i) The processor core used for demonstration is a 5-stage pipelined, single-width, open-source RISC-V core [2]. The L1.5 Cache design is agnostic to the pipeline depth and execution width (design method to support a super-scalar Out-of-Order (OoO) core is detailed in Sec. 3.3);

*Corresponding author: Shuai Zhao. Email: zhaosh56@mail.sysu.edu.cn.

DAC '24, June 23–27, 2024, San Francisco, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3657372

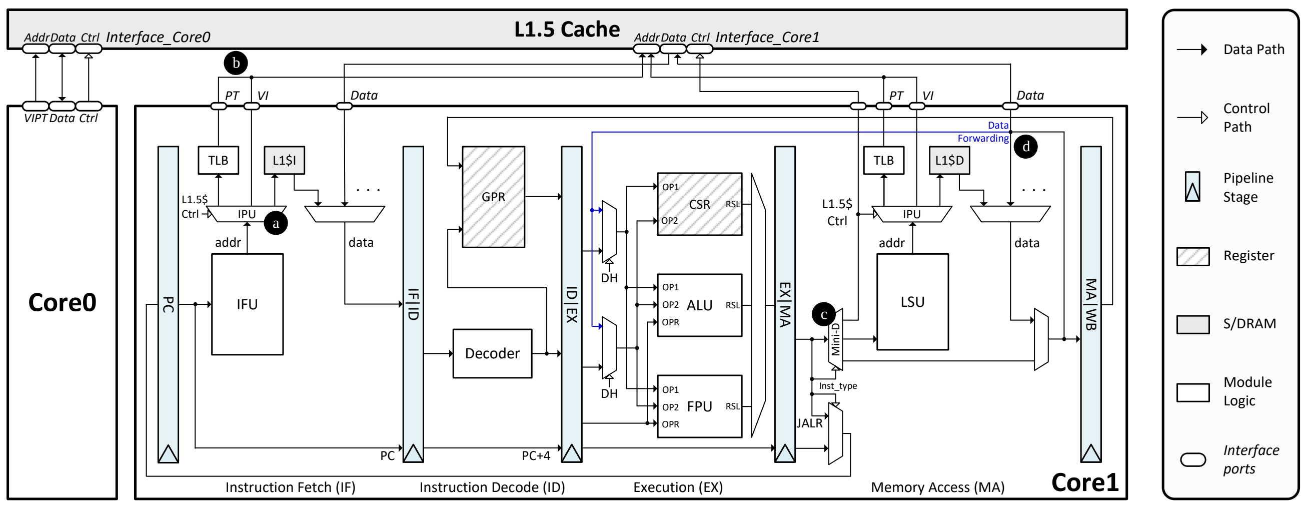


Figure 3: Integrating L1.5 Cache with 5-stage pipelined cores *(IPU: Inclusion Policy Unit; GPR: General-Purpose Register; CSR: Control Status Register; OP: Operand; OPR: Operator; RSL: Result; VI: Virtual Index: PT: Physical Tag; DH: Data Hazard).*

(ii) in alignment with modern processor design, the selected core incorporates a Translation Lookaside Buffer (TLB) and supports the full privilege levels stipulated by RISC-V, meaning that user applications always use virtual addresses for memory accesses.

2.1 Top-level Concepts

As discussed above, the inter-core cache interference in multi/many-core systems presents a major bottleneck for the timing performance of parallel real-time tasks with data dependency.

In coping with this issue, we present a VIPT and SINE cache (L1.5 Cache), deployed between the conventional L1 and L2 caches and shared across the cores in the same computing cluster (see Sec. 3). Different from the other shared caches, the proposed L1.5 Cache is tightly coupled to the cores' pipelines and offers flexibility for the software, e.g., the Operating System (OS), to (re)define characteristics at the *way level*. The reconfigurable characteristics of the cache ways include their ownership, global visibility to all cores in a cluster and inclusion policy (either inclusive or non-inclusive). A cache way is solely controlled and accessible by its designated owner, i.e., one of the cores in the cluster; however, it can be reassigned during execution. By setting the way to be "*globally visible*", it attains a read-only status, allowing the buffered contents to be seen by other cores in the same cluster. The benefits of the design are three-fold:

- The pipeline-coupled design offers an efficient "channel" for data sharing between the dependent tasks, accelerating the resolution of data dependency.
- The reconfigurations of the cache ways' ownership enable dynamic adjustments of cache capacity between parallel tasks, unblocking possible parallelism.
- The control over cache ways' visibility and inclusion policy enables precise and flexible data sharing, allowing optimisation of the L1.5 Cache's usage under different use scenarios.

With the L1.5 Cache, we establish a real SoC (Sec.2.2) and expand the conventional RISC-V ISA to offer dedicated interfaces for cache reconfigurations (Sec. 2.3). On the software side, we present a scheduling method (Sec. 4) to demonstrate the effectiveness of the L1.5 cache on a typical parallel task model with dependency, i.e., DAGs.

2.2 Integerating L1.5 Cache into a SoC

Fig. 3 depicts the integration of the L1.5 Cache into a multi/many-core SoC with 5-stage pipelined cores. Unlike the isolated L1 cache, the L1.5 Cache buffers both instructions and data collectively; thus, it is integrated into both the Instruction Fetch (IF) and Memory Access (MA) stages. To do this, we developed two Inclusion Policy Units (IPUs) using multiplexers and control registers, and connected them to the address ports of the Instruction Fetch Unit (IFU) and the Load Store Unit (LSU), respectively (see Fig. 3ⓐ). The IPU selectively routes memory accesses to the L1.5 Cache based on the configured inclusion policy and also combines the virtual index and physical tag (returned by the TLB) into the address port of the L1.5 Cache see Fig. 3ⓑ). With the IPUs, a demultiplexer is deployed at the IF and MA stages to route the read data back to the next stages.

We deployed a Mini-Decoder (Mini-D) at the MA stage to differentiate the conventional RISC-V and newly introduced L1.5 Cache ISA, constructing independent paths for each. The Mini-D directs the `load` and `store` instructions to the LSU and cache configuration instructions to the control port of the L1.5 Cache, with other instructions being passed straight to the WB pipeline stage (Fig. 3ⓒ).

Since reading data from the L1.5 Cache may induce data hazards for subsequent instructions, buffered at an earlier pipeline stage, we designed a forwarding channel, connecting the L1.5 Cache's data port and the Execution (EX) stage (Fig. 3ⓓ). This facilitates the direct passage of the dependent data, resolving hazards directly at the MA stage, rather than waiting until the data is written back.

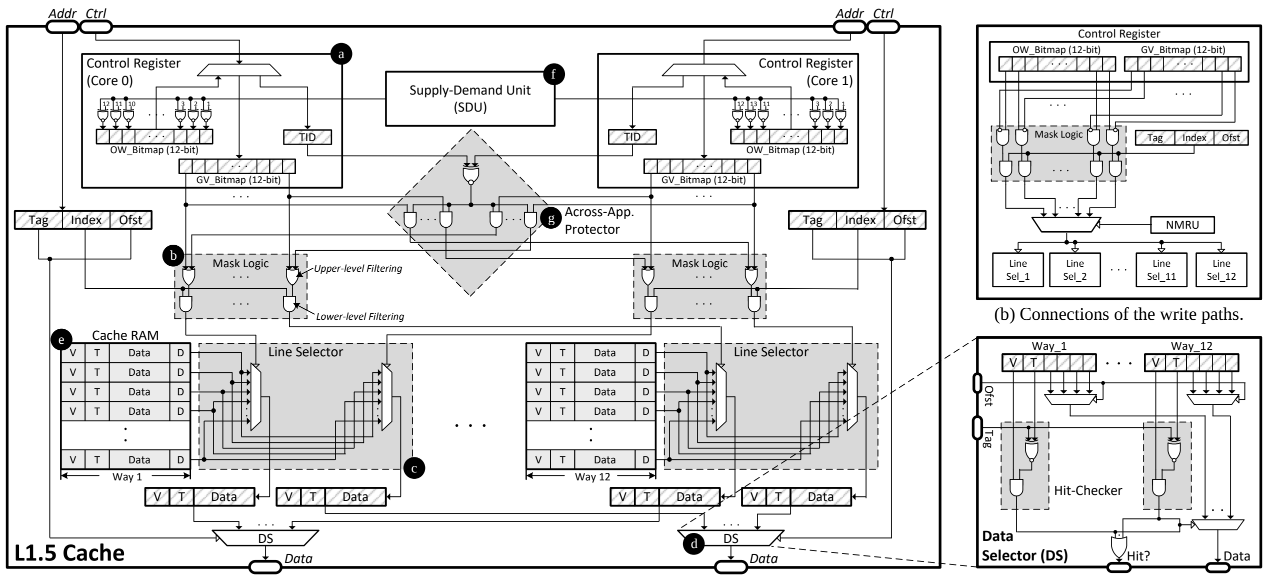
Table 1: New ISA for the L1.5 Cache reconfigurations *(Priv: 1 and 0 indicate the kernel and user modes, respectively).*

Instruction	Priv	Description
demand, rs1	1	Apply rs1 ways from L1.5 Cache.
`supply, rd`	0	Return the assigned ways in rd.
`gv_set, rs1`	0	Set owned ways' global visibility.
`gv_get, rd`	0	Return owned ways' global visibility.
`ip_set, rs1`	0	Set inclusion policy for all owned ways.

2.3 ISA Support

In coping with the reconfigurable characteristics, we developed a new ISA to abstract control interfaces for the software (Tab. 1). Specifically, we present an `ip_set()` instruction to configure the cache ways' inclusion policy, paired with `demand()` and `supply()` to apply the cache ways from the L1.5 Cache and check the cache ways have been properly assigned. Moreover, we present a pair of `gv_set()` and `gv_get()` instructions to set and check the global visibility of the cache ways owned by a core. To ensure the control efficiency, parameters delivered by `supply()`, `gv_set()`, and `gv_get()` are compacted using *bitmaps*. For instance, to set cache ways 2 and 7 to be globally visible, 0x42 is sent using `gv_set()`.

Given that the `demand()` can cause way contention between the cores, it is designed as a privileged instruction, executable only by an OS or a hypervisor with a comprehensive view of the system.



(a) Micro-architecture of L1.5 Cache *(OW: Ownership, GV: Global Visibility, TID: Task ID, V: Valid, T: Tag, D: Dirty)*.

(b) Connections of the write paths.

(c) Microarchitecture of data selector.

Figure 4: Microarchitecture of the L1.5 Cache, containing 12 cache ways and connecting two cores.

3 L1.5 Cache: Design

To enable dynamic sharing of L1.5 Cacheways among cores, we developed the L1.5 Cache microarchitecture using *bitmap-assisted mask circuits*. Specifically, we deploy a set of bitmap control registers, each of which is mapped 1-to-1 with a core, storing ownership (i.e., the allocated core) and global visibility of cache ways. When a memory request is dispatched to the L1.5 Cache's address port, the mask circuits filter and direct the memory request to the cache ways with the correct permissions based on the bitmap combinations.

3.1 The Microarchitecture

Fig. 4.(a) shows the L1.5 Cache microarchitecture, using the read paths as an example: ⓐ control registers record the access permissions of the cores; ⓑ mask logic filters requests and directs them to the corresponding selection logic (ⓓ and ⓔ), extracting the data from ⓒ cache ways; and ⓕ a Supply-Demand Unit (SDU) that distributes and maintains cache capacities between the cores.

Control registers. We divide the control registers into three (Fig. 4.(a)ⓐ): one for recording the Task ID (TID) of the core's running task, and the others for tracking the Ownership (OW) and Global Visibility (GV) of the cache ways assigned to the core. These registers are interfaced with the control port of the L1.5 Cache through a multiplexer for ISA accesses, where the modifications of the OW registers are fully handled by the SDU to minimise conflicts.

Mask logic. The mask logic links the control registers, the L1.5 Cache's address port and the cache ways, using a *dual-level filtering* mechanism (Fig. 4.(a)ⓑ): at the upper level, it combines all GV registers with the local OW register through OR-gates; at the lower level, the outputs from the OR-gates are connected to the index bits of the request using AND-gates. The connections direct requests to ways either owned by the associated core or globally shared.

Unlike read paths, write paths (Fig. 4 (b)) do not involve shared ways, thus require a different mask logic in the upper-level filtering. We deploy AND-gates to link the OW register with the NOT-gated GV register, selecting the ways owned by the core but not shared.

Cache ways and their selection logic. Cache ways were developed using SRAMs and organised using a *set-associative* architecture [9]. Each line is partitioned into four: valid, tag, data and dirty (Fig. 4.(a)ⓒ). The valid bitfield indicates the validity of the line; the tag and data bitfields store the tag and data of the mapped memory block; and the dirty bit reveals if the line requires coherence.

With the cache ways, we designed Line Selectors (LSs) and Data Selectors (DSs) to retrieve the buffered data. The LS directs all lines of a cache way to the data ports of the multiplexers and routes the outputs to the DSs (Fig. 4.(a)ⓓ), where the multiplexers' control ports are connected to the mask logic's outputs. Each DS is linked with a core, validating if a request meets a cache-hit (Fig. 4(c)). To do this, the DS uses a set of latches to buffer the outputs from the LSs. Each LS is connected by a hit-checker, comprising an XNOR-gate and an AND-gate. The XNOR-gate connects to the latch's tag, assessing the cache-hit status, while the AND-gate connects to the XNOR-gate's output and the valid bitfield, checking its validity.

Supply-Demand Unit (SDU). SDU (Fig.5) receives a `demand()` from cores and `supply()` ways with its best efforts. The SDU design has a set of Supply-Demand (SD) registers, comparators and a Way Allocator (Walloc). The SD registers (Fig. 5ⓐ) are associated with a core, buffering the core ID, the number of ways being demanded via a Demand (D) register and those currently supplied via a Supply (S) register. A comparator (Fig. 5ⓑ), comprising a subtractor and an XOR-gate, links the S and D registers. The subtractor computes the gap between S and D registers, while the XOR-gate finds a mismatch. When a mismatch is detected, the gap value is sent to the Walloc joined with the core ID. The Walloc (Fig. 5ⓒ) was designed using a FSM alongside a register bank and register controller. The register bank acts as a "shadow" of the ways' ownership, allowing the FSM to alter the ownership by its controller. When extra cache ways are allocated, Walloc writes to slots that are unoccupied (marked as N/U). Conversely, when the number of cache ways is reduced, Walloc marks the relevant slots with N/U. After updating the register bank, the new value is sent to the corresponding control register using a bitmap, while the Walloc updates the S register.

3.2 Mitigating Cross-application Invasion

Different applications might be executed in a single cluster, each involving a unique mapping of virtual-to-physical addresses. Thus, cross-application cache sharing is not allowed [9]. To enforce this, a protector, employing XNOR- and AND-gate to govern the connectivity of the control registers (Fig. 4(a)ⓖ). The XNOR-gate connects the TIDs across the control registers, checking if the same application is being executed. The XNOR-gate's output is interfaced with the GV registers using AND-gates, before being sent to the mask logic, preventing cache sharing between different applications.

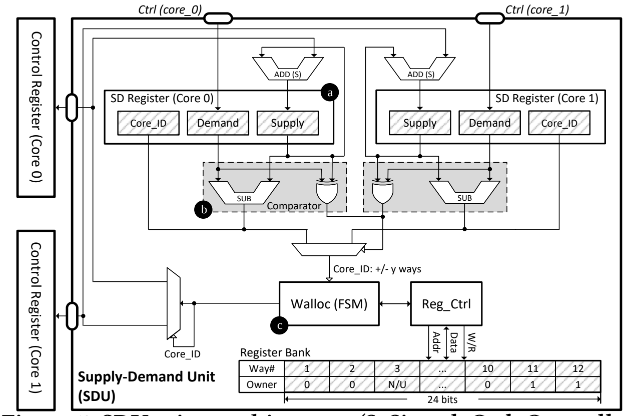


Figure 5: SDU microarchitecture *(S: Signed, Ctrl: Controller).*

3.3 Supporting Instruction-level Parallelism

The proposed L1.5 Cache is also compatible with superscalar OoO cores, where multiple memory requests may be dispatched in one cycle. To facilitate instruction-level parallelism, several modifications are necessary. At the interface, additional address and data ports are required to interface with head entries of Load and Store Queues (LSQs) to handle simultaneously dispatched requests. Prior to the mask logic, an extra buffer should be instantiated to temporarily store and prioritise the in-flight requests.

4 Exploiting L1.5 Cache in DAG Scheduling

This section presents a DAG schedule that utilises the L1.5 Cache to reduce the communication cost and the programming model.

4.1 System Model

Task Model. We consider a recurrent DAG task $\tau_i = \{V_i, E_i, T_i, D_i\}$, in which V_i denotes a set of nodes, E_i is a set of edges, T_i is the period of τ_i and D_i is the deadline with $D_i \leq T_i$. For a node v_j, C_j denotes the worst-case computation time (WCET) and P_j is the priority. Function $pre(v_j)$ returns nodes that are connected to v_j (i.e., the predecessors) while $suc(v_j)$ gives the successors connecting from v_j. The volume of data produced by v_j (consumed by $suc(v_j)$) is denoted by δ_j, and can be obtained using profiling tools, e.g., Valgrind [1]. An edge $e_{j,k}$ connecting v_j and v_k has a communication cost $\mu_{j,k}$. A path is a node sequence in which any two consecutive nodes are connected with an edge. The length of the longest path that contains v_j is denoted by λ_j. As with [8], τ_i has one source node v_{src} and one sink node v_{sin}. Nodes are scheduled by a non-preemptive fixed-priority scheduler with the work-conserving scheme [8].

Cache Model. The system contains a typical two-level (L1 and L2) cache hierarchy along with the L1.5 Cache described in Sec. 2 (see Fig. 2). The L1.5 Cache has in total ζ cache ways and each way has a size of κ. An L1.5 Cache way can be configured as *local* or *global*. A local way is dedicated to a node with full control (i.e., read and write), whereas a global way can be read by all nodes in τ_i. Each way can be configured as non-inclusive (by default) or inclusive, and has a unique owner node, i.e., the core designated to the node.

Timing Model. The Execution Time Model (ETM) in [15] is applied to compute the speed-up of the communication cost of an edge $e_{j,k}$ given n L1.5 Cache ways. As described in [15], for a dedicated cache without any inter-core interference, the ETM can be effectively constructed as $ET(e_{j,k}, n) = \mu_{j,k} \times (1 - \alpha_{j,k} \times \frac{n}{\lceil \delta_j/\kappa \rceil})$, in which $\lceil \delta_j/\kappa \rceil$ is the number of ways required for storing the dependent data produced by v_j and $\alpha_{j,k}$ is the speed-up ratio of $e_{j,k}$. Details of the ETM construction with verified feasibility can be found in [15].

4.2 DAG Scheduling with L1.5 Cache

L1.5 Cache Configuration. Each node v_j is assigned a number of *local* or *global* L1.5 Cache ways. The local ways allow v_j to store dependent data for $suc(v_j)$, whereas the global ones contain data required by v_j. The local ways are configured as global once v_j is finished, making the dependent data visible to all $suc(v_j)$. Fig. 6 illustrates the cache configuration of an example DAG task. For v_1, a set of local ways are assigned to store its dependent data. As $pre(v_1) = \emptyset$, v_1 has no global ways. Once v_1 is finished, its local ways are set as global for v_2 (the first successor of v_1), and are shared by v_2, v_3 and v_4. Then, local ways are also assigned to these nodes. The same configuration applies to the following nodes, in which v_7 is only assigned with global ways as $suc(v_7) = \emptyset$.

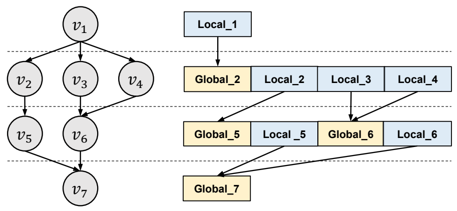


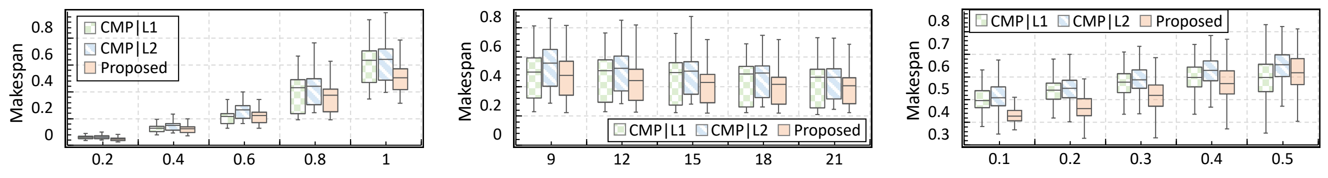
Figure 6: The L1.5 Cache configuration of an example DAG.

Algorithm 1: DAG scheduling with L1.5 Cache.

```
1  $Q = v_{src}$;   $\Omega = \emptyset$;   $pri = |V_i|$;
2  while $Q \neq \emptyset$ do
3      for $\omega_x \in \Omega$ do
4          /* L1.5 global cache ways configuration */
5          if $\omega_x$ is local then
6              $\omega_x.type = global$;   $\omega_x.owner = suc(\omega_x.owner).first()$;
7          else
8              $\Omega = \Omega \setminus \omega_x$;
9          end
10     end
11     for $v_j \in Q$ do
12         $v_j = \underset{j}{argmax}\{\lambda_j \mid \forall v_j \in Q\}$;
13         /* local L1.5 cache ways configuration */
14         if $\Omega.size < \zeta$ then
15             $\omega_x.size = F(v_j, \Omega, \zeta)$;   $\omega_x.owner = v_j$;   $\Omega = \Omega \cup \omega_x$;
16         end
17         /* node priority assignment */
18         $P_j = pri$;   $pri = pri - 1$;   $Q = Q \setminus v_j$;
19     end
20     update $\lambda_j, \forall v_j \in V$ via dynamic programming;
21     update $Q$ based on the precedence constraint of examined nodes;
22 end
```

Scheduling with L1.5 Cache. With the cache configuration in Fig. 6, Alg. 1 presents the proposed DAG schedule that exploits the L1.5 Cache. The proposed method takes a DAG task (τ_i) and the number of ways in the L1.5 Cache (ζ) as the input, and produces the L1.5 Cache configuration and priority of each node in τ_i. Essentially, the method follows the principle that always assigns dedicated L1.5 Cache ways and higher priority to nodes in the longer path, so that the DAG makespan can be effectively reduced. Notation Q denotes the set of currently-examined nodes, Ω contains the currently-allocated L1.5 Cache ways, pri is the current priority level. For simplicity, we let ω_x denote a group of L1.5 Cache ways assigned to a node with three attributes: (i) number of ways ($\omega_x.size$), (ii) type ($\omega_x.type = \{local, global\}$), and (iii) owner node ($\omega_x.owner$).

The algorithm starts from the source node, i.e., $Q = \{v_{src}\}$. In each iteration, the algorithm first sets the local ways in Ω as global (if any, owned by nodes in Q of the previous iteration), and updates the ownership accordingly, allowing nodes in Q to access the dependent data from their predecessors (lines 5-7). The global ways in Ω are freed as they are no longer required (line 8). Then, starting from the node v_j with the highest λ_j in Q (line 12), the algorithm determines the number of local L1.5 Cache ways (lines 14-16) and the priority (line 18) for v_j. Function $F(v_j, \Omega, \zeta) = \min\{\lceil \delta_j/\kappa \rceil, (\zeta - \sum_{\omega_x \in \Omega} \omega_x.size)\}$ computes the number of L1.5 Cache ways supplied to v_j, in which $(\zeta - \sum_{\omega_x \in \Omega} \omega_x.size)$ is the



(a) $p = 15$, $cpr = 0.28$ and varied U_i. (b) $U = 0.8$, $cpr = 0.3$ and varied p. (c) $U = 0.8$, $p = 15$ and varied cpr.

Figure 7: Comparison of normalised makespan under varied U_i, p and cpr of a DAG task.

number of ways that can be allocated. Then, a priority is assigned to v_j following the principle of the node in the longer path first.

After nodes in Q are examined, λ_j of all nodes is updated based on $ET(e_{j,k}, n)$, given newly-allocated cache ways (line 20). By doing so, the algorithm keeps tracking long paths in τ_i and prioritises the reduction of their communication cost and execution, effectively reducing the DAG makespan. Finally, Q is updated with nodes for the next iteration (line 21). The algorithm finishes when $Q = \emptyset$, and returns L1.5 Cache configuration and priority for nodes in τ_i. Note, the proposed method does not undermine the predictability, as the inter-core interference is eliminated in the L1.5 Cache. Existing analysis (e.g., the one in [8]) can be applied to provide safe timing bounds, with minor modifications for communication cost on edges. The time complexity of Alg. 1 is cubic, which contains at most $|V_i|$ iterations with a quadratic complexity for updating λ_j and Q [8].

4.3 Programming Model

We now present the programming model that supports the proposed I/O allocation and scheduling method with the L1.5 Cache. Alg. 1 specifies the number of local and global L1.5 Cache ways of each node. For a node v_j, the L1.5 Cache configuration is conducted before it is dispatched by the scheduler, e.g., during the context switch. For the local ways, `demand()` (see the ISA in Tab. 1) is invoked given the number of ways assigned to v_j. Then, `ip_set()` is used to set the ways as inclusive, so that the dependent data produced by v_j can be written to the L1.5 Cache through the L1 cache. For the global cache ways, `supply()` is invoked to obtain the local ways for v_j's predecessors, along with `gv_set()`, which sets the global visibility of these ways, enabling read-only accesses of nodes that require the same data.

5 Evaluation

Experimental platform. We built 8/16-core SoCs on the AMD Alveo U280. We implemented the cores based on the Rocket [2], an open-source RISC-V core. We configured the cores with a 5-stage pipelined, in-order dispatch, with microarchitectural changes to support the new ISA (Sec. 2.3). Each core was equipped with an independent 4KB I$ and D$ (1 - 2 cycles latency). We organised these cores into clusters, each comprising four cores sharing a L1.5 Cache. The L1.5 Cache was implemented using Bluespec SystemVerilog under the guidelines in Sec. 3, featuring 16 ways (each containing 2KB, 2 - 8 cycle latency). We integrated the computing clusters, the L1.5 Cache, L2 cache (512 KB, 15 - 25 cycles latency) and external memory (4GB@800Mhz) as an SoC using the approach described in Sec. 2.2. The hardware was synthesised, implemented and routed using Vivado (v.2021.1). FreeRTOS (v.10.4) was used as the OS kernel across all cores (see Sec. 2.3). All software (OS kernels, drivers and user applications) was compiled using a RISC-V GNU toolchain.

We built three baseline systems as **Comparators (CMPs)**. Both **CMP|L1** and **CMP|L2** are legacy real-time systems developed upon the above hardware without the L1.5 Cache. **CMP|Shared-L1** [10] is a system with a shared L1 cache, using a heuristic for capacity allocation. For the CMPs, the L1 and L2 capacity was increased to ensure that the total cache size was equivalent across all systems.

Table 2: Comparison of the normalised worst-case makespan.

U_i	CMP [15]	Prop.	p	CMP [15]	Prop.	cpr	CMP [15]	Prop.
0.2	0.095	0.061	**9**	0.446	0.35	**0.1**	0.378	0.296
0.4	0.189	0.137	**12**	0.405	0.313	**0.2**	0.40	0.310
0.6	0.283	0.214	**15**	0.379	0.294	**0.3**	0.427	0.338
0.8	0.378	0.292	**18**	0.362	0.281	**0.4**	0.464	0.380
1.0	0.472	0.372	**21**	0.348	0.275	**0.5**	0.509	0.427

The pseudo-LRU is applied for all caches with the same memory model [11], simulating 1.2 GHz systems at 30 MHz.

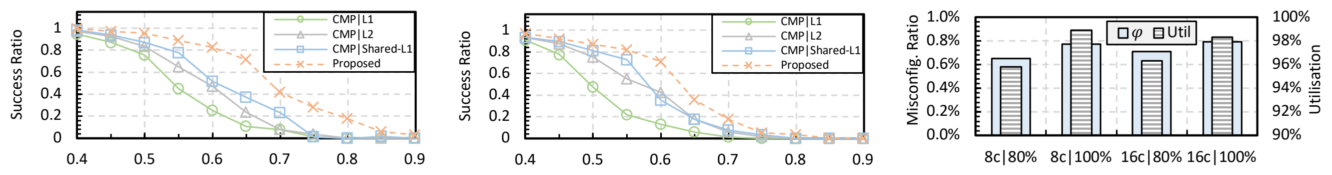
5.1 Makespan Comparison

Experimental setup. We evaluated the proposed schedule with the L1.5 Cache against the SOTA in [15], which provides a DAG scheduling solution that benefits from the caches in CMP|L1 and CMP|L2. The evaluation was conducted on the simulator developed in [15] with synthetic DAGs. A DAG task was generated as follows: the number of layers was randomly decided in [5, 10], and the number of nodes in each layer was decided in [2, p] ($p = 15$ by default). A node had a probability of 20% to connect with every node in the previous layer. The period T_i was randomly generated in [1,1440] units of time with $D_i = T_i$. The workload $W_i = U_i \times T_i$ was computed given a utilisation U_i. The WCETs of nodes were then generated uniformly based on W_i. A parameter *critical path ratio* (cpr) was used to control the proportion of the longest path in τ_i, e.g., $cpr = 20\%$ means the length of the longest path is $W_i \times 20\%$. The ratio between the sum of communication cost (denoted $\sum_\mu$) and W_i was 0.5, with the communication cost of each edge generated in $[1, \sum_\mu / |E_i| \times 2]$. Each edge has an ETM with $\alpha_{j,k}$ generated in (0, 0.7], i.e., with a maximum speed-up of 70% from the L1.5 Cache.

Results. The proposed DAG schedule with the L1.5 Cache outperformed the SOTA in terms of the DAG makespan. This is observed from Fig. 7, which shows the average makespan (normalised by the highest value observed) of the first 10 instances of 500 DAGs. Our method provided the lowest makespan in general, e.g., outperformed CMP|L1 and CMP|L2 by 11.1% and 22.9% on average in Fig. 7(a), respectively. In particular, it showed a strong performance with a relatively high U_i or a low cpr, e.g., $U_i \geq 0.8$ or $cpr \leq 0.3$. In such cases, the traditional cache can hardly speed up the nodes due to high WCET. Hence, the communication cost becomes the bottleneck for reducing DAG makespan, which was decreased using L1.5 Cache. Tab. 2 shows the normalised worst-case makespan of competing methods. Our method outperformed the CMP by 26.3%, 22.1%, and 19.9% on average with the varied U_i, p and cpr, respectively. Notably, the traditional cache requires a warm-up phase to speed up the node execution, leading to a high worst-case makespan. By contrast, with the L1.5 Cache, the dependent data that a node requires is available in the cache each time it is released, hence, achieves a significantly lower worst-case DAG makespan.

5.2 Case Study

Experimental setup. We examined the real-time performance of the systems using a case study. We configured all systems with 8/16 cores and then executed the Parsec 3.0 benchmark (multi-thread version) [3] with the `simsmall` input. We slightly modified the workloads by introducing precedence constraints and data flow



(a) Success ratio (8 cores, 2 clusters). (b) Success ratio (16 cores, 4 clusters). (c) Side-effects analysis.

Figure 8: Case study. In Fig. 8(a) and 8(b), x-axis: target utilisation; In Fig. 8(c): xc$|y$%: an SoC with x cores and y% utilisation.

dependency between the threads to form DAG tasks. Before run-time, the raw data used by the tasks was generated and stored in the memory. At run-time, the cores fetched the raw data, executed the tasks, and then sent the calculated results back to the memory. Each task had a randomly defined period and implicit deadline. We executed the examined systems 200 times under varying target utilisation [40%, 90%], at intervals of 5%. The dependent data shared between the nodes was synthetically generated with a random size [2KB, 16KB]. For fair comparisons, we ensured the dependent data and timing parameters in each trial were identical. We examined the real-time performance using the *success ratio*, which recorded the percentage of trials that were executed without deadline misses.
Results. Deploying the L1.5 Cache in conventional multi/many-core SoCs significantly improved the systems' real-time performance. As shown in Fig. 8(a) and 8(b), when systems were configured with the same settings (core number and target utilisation), our proposed solution consistently outperformed CMPs in terms of success ratios [5%, 40%]. Such improvements benefited from deploying the L1.5 Cache (Sce. 2) and suitable configurations (Sec. 4) for the DAG tasks, effectively increasing the parallelisms of execution and expediting the sharing of interdependent data.

5.3 Side-effects Analysis

The L1.5 Cache introduces a range of valuable features previously unseen in L1 and L2 caches. Therefore, it is important to understand the effectiveness and potential gaps in the L1.5 Cache design, especially during time periods when the system is busy. The potential side-effects brought by the L1.5 Cache are twofold: (i) decreased cache utilisation due to the additional cache management; and (ii) cache configuration delays caused by contentions between cores.
Experimental setup. We adopted the same experimental setup and methods as in Sec. 5.2, with only the proposed system being executed. To replicate a high-demand scenario, we configured 8/16-core systems with 80% and 100% utilisation. We deployed a cycle-accurate monitor to trace the cores and L1.5 Cache recording (i) the utilisation of the L1.5 Cache and (ii) the configuration latencies. The utilisation was calculated by the percentage of the cache ways that had been assigned. The latency was assessed by determining the percentage φ of task executions that occurred with an unexpected setting. For example, if a task executes in 100ms with the correct settings and 1ms with the incorrect settings, φ_i would be 1%.
Results. In busy periods, the L1.5 Cache was effectively utilised and the miss-configuration latency was less than 1% of task executions. In Fig. 8(c), with the systems configured at 80% utilisation, the average L1.5 Cache utilisation surpassed 95%. When increasing the system's utilisation to 100%, the L1.5 Cache utilisation exceeded 98%, thereby ensuring the stability of the systems' throughput. Across all experimental setups, φ consistently remained below 1%. Nevertheless, when the system was set up with higher utilisation, a minor increase in φ was observed. This is chiefly attributed to the DSU's constraint of configuring only one cache way at a time.

5.4 Hardware Overhead

Experimental setup. We conducted a physical implementation of a 16-core SoC (at 400 Mhz) with the L1.5 Cache (32KB, 8 ways per computing cluster) and with only the L1 Cache (8KB, 2 ways per core), which was carried out at the post-layout stage using Synopsys 28nm Generic PDKs [6]. The RTL was synthesised using the Synopsys Design Compiler (v2022.12). The synthesised netlist was placed and routed with the Synopsys IC Compiler 2 (v2022.12).
Results. The SoC has a reported area of 2.757mm^2, with each cluster contributing 0.574mm^2. Within an individual cluster, the four processors occupy 0.359mm^2, and the integration of the new ISA entails an area overhead of approximately 0.001mm^2 per core. In comparison, the SoC designed using the conventional L1 cache with the same capacity yields a reduced total area of 2.604mm^2, due to its relatively simple microarchitecture. In summary, developing a 16-core SoC with the L1.5 Cache results in an increased area consumption of 0.153mm^2, representing 5.88% of the SoC's area.

6 Conclusion

This paper presented a VIPT and SINE L1.5 Cache that provides configurable and fine-grained data-sharing capabilities for parallel tasks with data dependency. Focusing on a mainstream parallel task model, i.e., DAGs, a scheduling method was proposed that exploits the L1.5 Cache to reduce the commutation cost between nodes, hence, the DAG makespan. As a complete solution, a systematic full-stack framework was constructed from the SoC and the ISA to the programming model. Experimental results demonstrated the effectiveness of the proposed solution over the SOTA method in terms of timing performance of DAG tasks with negligible overheads.

Acknowledgment

The authors would like to thank the anonymous reviewers for their insightful and helpful feedback. This work is supported by National Key Research and Development Program under Grant 2023YFB4503703, National Natural Science Foundation of China under Grant 62302533, Guangzhou Fundamental Research Funds under Grant SL2023A04J00996, the Fundamental Research Funds for the Central Universities, and U.S. National Science Foundation under Grant CNS–1948457.

References

[1] Valgrind. https://valgrind.org/. Accessed November. 7, 2023.
[2] Krste Asanovic et al. 2016. The rocket chip generator. *University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* (2016).
[3] Christian Bienia et al. 2008. The PARSEC benchmark suite: Characterization and architectural implications. In *PACT*.
[4] Jichuan Chang et al. 2007. Cooperative cache partitioning for chip multiprocessors. In *ACM International Conference on Supercomputing*.
[5] Peng Chen et al. 2019. Timing-Anomaly Free Dynamic Scheduling of Conditional DAG Tasks on Multi-Core Systems. *ACM TECS* (2019).
[6] R Goldman et al. 2013. 32/28nm educational design kit. In *PrimeAsia*.
[7] N. Guan et al. 2009. Cache-aware scheduling and analysis for multicores. In *ACM international conference on Embedded software*.
[8] Qingqiang He et al. 2019. Intra-task priority assignment in real-time scheduling of DAG tasks on multi-cores. *IEEE TPDS* (2019).
[9] John L Hennessy et al. 2011. *Computer architecture: a quantitative approach*.
[10] Z Jiang et al. 2024. Hopscotch: A Hardware-Software co-Design for Efficient Cache Resizing on Multi-core SoCs. *IEEE TPDS* (2024).
[11] Sagar Karandikar et al. 2018. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *ACM/IEEE ISCA*.
[12] David Blair Kirk. 1989. SMART (strategic memory allocation for real-time) cache design. In *IEEE RTSS*.
[13] Jinghao Sun et al. 2023. Real-Time Scheduling of Autonomous Driving System with Guaranteed Timing Correctness. In *IEEE RTAS*.
[14] Haluk Topcuoglu et al. 2002. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE TPDS* (2002).
[15] Shuai Zhao et al. 2023. Cache-Aware Allocation of Parallel Jobs on Multi-cores based on Learned Recency. In *ACM RTNS*.