Ditty: Directory-based Cache Coherence for Multicore Safety-critical Systems

Zhuanhao Wu, Marat Bekmyrza, Nachiket Kapre, Hiren Patel
University of Waterloo, Waterloo, Ontario, Canada
{zhuanhao.wu, marat.bekmyrza, nachiket, hiren.patel}@uwaterloo.ca

***Abstract*—Ditty is a predictable directory-based cache coherence mechanism for multicore safety-critical systems that guarantees a worst-case latency (WCL) on data accesses. Prior approaches for predictable cache coherence use a shared snooping bus to interconnect cores. This restricts the number of cores in the multicore to typically four or eight due to scalability concerns. Ditty takes a first step towards a scalable cache coherence mechanism that is predictable and one that can support a larger number of cores. In designing Ditty, we propose a coherence protocol and micro-architecture additions to deliver a WCL bound that is lower than a naive approach. Our WCL analysis reveals that the resulting bounds are comparable to state-of-the-art bus-based predictable coherence approaches. We prototype Ditty in hardware and empirically evaluate it on an FPGA. Our evaluation shows the observed WCL is within computed WCL bound for both the synthetic and SPLASH-3 benchmarks. We release our implementation to the public domain.**

I. Introduction

Predictable cache coherence approaches for safety-critical systems (SCS) promise high performance when accessing data by enabling private caching data while ensuring worst-case latency (WCL) bounds on accesses [1], [2]. Example domains that can benefit from these approaches include avionics, automotive, and space [3]. Existing state-of-the-art approaches that propose predictable cache coherence [1]–[5] offer solutions that use a shared snooping bus. These are apt solutions for multicores with a low core count such as the NXP T2080 that are equipped with cache coherence [6]. However, it is well understood that a shared snooping bus makes scaling to a large number of cores impractical [7]. Nonetheless, addressing this drawback is imperative to meet today's rising need for additional functionalities, further consolidation of existing functionalities, delivering higher performance, and reducing cost, size, weight, area, and power. Consider the Kalray MPPA 3 [8] that has an 80-core architecture, where clusters are connected through network-on-chips and L2 caches are partitioned across clusters. State-of-the-art approaches cannot be directly applied to such multicore architectures. Therefore, developing a solution that enables predictable cache coherence for larger core counts is paramount for future systems.

Ditty presents one such solution: a directory-based cache coherence approach for SCS that scales to a larger number of cores, guarantees a WCL, and delivers high performance via caching. It is well-known that directory-based cache coherence promotes scalability [9]; however, to the best of our knowledge, there is no prior effort in developing a predictable directory-based cache coherence mechanism. With Ditty, we take the first step in doing so. In designing Ditty, we encountered and addressed several important challenges. First, analyses in prior works [1], [2], [4] do not explicitly handle the effect of back-invalidations (a replacement in the LLC requiring evictions of copies that are privately cached) on the WCL when including an inclusive LLC. Ditty addresses this challenge by explicitly accounting for this in its analysis. Second, naively interconnecting cores using a predictable interconnect such as HopliteRT [10] and employing a general purpose cache coherence protocol [11] may result in a prohibitively large WCL. For this, Ditty methodically identifies scenarios that indeed result in a large WCL, and proposes coherence protocol and micro-architectural additions to lower the WCL.

To summarize, our main contributions are as follows.

- We propose a predictable directory-based cache coherence approach for safety-critical systems called Ditty. Ditty employs HopliteRT [10] as its predictable interconnect; however, any predictable interconnect would suffice. Ditty includes carefully considered micro-architectural additions to the L2 cache controller and the directory. In addition, Ditty extends the Modified-Shared-Invalid (MSI) cache coherence protocol to complement the micro-architectural additions.
- We present a timing analysis for Ditty that computes the WCL a memory request can experience. In this analysis, we identify the critical instance, which centers on the eviction of cache lines in the LLC.
- We prototype Ditty hardware, and deploy it on an FPGA [1].

II. Background and Related Works

A. Hardware cache coherence mechanisms

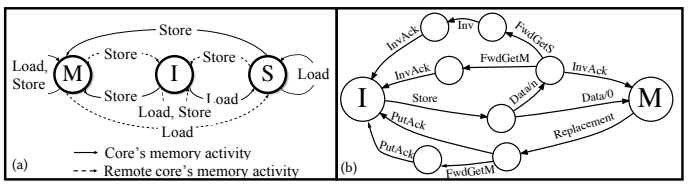

Fig. 1: (a) MSI cache coherence protocol. (b) The detailed transitions between M and I.

A hardware cache coherence protocol (CCP) dictates the operations on data performed by agents (cores and shared

[1]Code available at https://github.com/caesr-uwaterloo/Ditty

978-3-9819263-7-8/DATE23/© 2023 EDAA

memory) in the memory hierarchy. CCPs ensure that all cores observe the same value at a given (logical) time. Figure 1a shows the MSI cache coherence protocol [11]. In MSI, each cache line is in one of the following states: Modified (M) state, Shared (S) state, or Invalid (I) state. A cache line in M state means that the contents have been modified by the core, and it has read and write access permissions on that cache line. For shared data, only one core can have a cache line in M state. S state denotes that a core has read access permission, and the data contents are unchanged. Multiple cores (called sharers) can have a cache line for the same address in S to allow read-only access to shared data. A cache line in I state is not available in the core's private cache or its data contents are not up-to-date. M, S, and I states are also called *stable states*.

Practical implementations of a CCP use *transient states* to improve performance [11]. These transient states identify that a cache line is transitioning between two stable states. Transient states are essential for overlapping the execution of memory requests. As an example, Figure 1b shows the transient states of the transitions between M and I at the core. A core caching a cache line in M needs to invalidate it due to limited set-associativity to serve new requests, causing *replacement* on the cache line. This core transitions from M into a transient state MI, meaning that the cache line is transitioning from M to I. Once the core receives an acknowledgment of the replacement being done (PutAck), it transitions into I. MI allows other cores to access this cache line, and for the core performing the replacement to respond.

B. Related works

Directory-based coherence mechanisms are common in general purpose multicore platforms [12], [13]. However, there are no prior efforts in developing a predictable directory-based cache coherence mechanism that can be used in safety-critical systems. Ditty, to our knowledge, is the first to do so.

Most of the recent research activity focused on cache coherence mechanisms for snooping bus-based multicores for safety-critical systems [1]–[5]. PMSI [4] utilized a shared command and data bus for communication where an LLC was not present resulting in a WCL that was quadratic in terms of the number of cores. PMSI* [1] proposed coherence protocol changes that lower the WCL to linear. PISCOT [2] separated the request bus and response bus, where the request bus deploys a time-division multiplexing arbitration and the response bus features a first-come-first-serve arbiter to boost performance. PISCOT's WCL is linear to the number of cores. Authors in [5] enable predictable coherent data sharing with LLC through the use of a separate invalidation bus and the use of a write-back buffer at the LLC. Authors in [14] deploy COTS coherence protocols on a shared bus deploying predictable arbitration scheme such as TDM, where a direct cache-to-cache interconnect is used to ensure transmission of data within a single TDM slot, achieving a linear WCL. Ditty does not use a shared snooping bus. Instead, it uses a directory-based coherence protocol with a predictable interconnect for transmitting both the command and data, and includes an inclusive LLC.

III. System Model

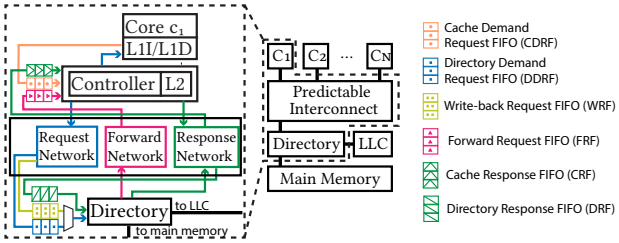

Fig. 2: Ditty system model.

Memory hierarchy. Our system model has N cores, a 3-level inclusive cache hierarchy, and a directory connected to a main memory as shown in Figure 2. Caches transfer data at 16-byte cache line granularity. L1 and L2 caches are private to the cores and the LLC is shared among all cores. All caches use the least-recently-used replacement policy. A directory tracks cache lines privately cached in L2 caches, maintaining coherent memory accesses across all cores by being the ordering point. Existing multicore platforms such as ARM DynamIQ DSU and AMD Opteron platforms use a similar setup [13], [15]. We use MSI to manage coherence between the cores [4]. Table I shows the messages used for communication by the L2 and the directory. Our analysis focuses on characterizing the WCL when the core issues a request that misses in the L1 and must access L2 to obtain the latency, that is, the worst-case miss latency of L1.

Interconnect. We require a predictable interconnect to have the following properties: (1) point-to-point communication among connected end-points with three networks, and (2) a worst-case communication latency (WCCL) when sending messages or data between two end points. General purpose multicores [12], [16] satisfy (1) by using three networks for different types of messages: request, forward, and response. This delivers high performance and ensures correctness [11]. For (2), the WCCL is essential for safety-critical systems and recent efforts include [10], [17]. Note, for transmission of point-to-point messages in (1), Ditty supports interconnects with point-to-point ordering properties [11], [16] and those without by using acknowledgment messages, making the protocol applicable to point-to-point ordered and unordered interconnects. For our evaluation, we use HopliteRT [10], but Ditty works with any interconnect with the two properties.

FIFOs. Both L2 and the directory use FIFOs to buffer incoming and outgoing messages as shown in Figure 2. The L2 cache controller has three input FIFOs: There are three input FIFOs at the directory as well: write-back request (WRF), directory demand request (DDRF), and response (DRF). We employ a predictable arbitration between CDRF and FRF. The CDRF for L2 caches is of size 1. All other FIFOs for L2 caches and the directory are of size N.

► *Core and L2 cache operations.* A core requests data from the memory using a Load and writes data to the memory using a Store. These requests access the L1 cache first. If these requests hit in the L1 cache then the data is returned to the core for a

TABLE I: Coherence messages in L2 and LLC

Message	Description	FIFO
Read	Get a shared copy of a cache line	DDRF
Write	Get an exclusive copy of a cache line	DDRF
PutS	Write back a shared copy of a cache line	WRF
PutM	Write back an exclusive copy of a cache line	WRF
FwdGetS	Forward a shared copy of a cache line to another core	FRF
FwdGetM	Forward an exclusive copy of a cache line to another core	FRF
Inv	Invalidate a shared copy of a cache line	FRF
PutAck	Acknowledgement of a write-back request	CRF
InvAck	Acknowledgement of an Inv	CRF, DRF
FwdAck	Acknowledgement of FwdGetS/FwdGetM	CRF, DRF
Data	Data response	CRF, DRF

Load, and written to the L1 cache for a Store. If these requests miss in the L1 cache then L2 receives the Load and Store requests on the core's behalf. As in prior works [15], [16], [18], cache coherence operates between the L2 and the LLC. Hence, we focus our discussion on the interaction between the L2s, directory, and LLC. If Load or Store hits in the L2 cache, the requested data is returned to the core via the L1 (following inclusivity). Otherwise, the L2 cache controller issues *demand requests*, a Read for a Load or a Write for a Store, to the directory. Read requests the data to be read by the core, and Write requests the data that the core plans to modify. Before sending a Read or a Write, the L2 cache may not have a vacant entry for the requested data and must evict a *victim* cache line. For this, the L2 cache controller issues a PutS (clean copy of data) or a PutM (dirty copy of data) to relinquish its private copy of the victim cache line. We refer to PutS or PutM requests as *write-back requests*. Hence, we say that a core issues a request or receives a response means that the core's L2 cache controller issues a request or receives a response. Table I shows the usage of FIFOs for each of the coherence messages. We assume a core can only have an outstanding demand request.

▶ *Directory operations.* Upon receiving a demand request from the L2 cache controller, the directory orchestrates the necessary communication and sends the requested cache line to the L2 while maintaining the coherence of data between all cores. The directory tracks the owner and a list of sharers of each privately cached cache line. Since L2 is inclusive, only the lines in L2 are tracked. Incoming demand requests and write-back requests are buffered in DDRF and WRF at the directory. The directory prioritizes write-back requests over demand requests. On receiving a PutS (write-back of a clean cache line) request for a cache line A from a core c_{ua}, the directory removes c_{ua} from the sharer list of A so that the directory does not have to invalidate the copy in c_{ua} when a later request modifies A. For a PutM (write-back of a modified cache line), the directory marks that A is no longer exclusively cached by c_{ua}. The PutM carries data modified (dirty) by c_{ua}; hence, the directory updates the LLC with the most up-to-date data.

IV. Directory-based cache coherence mechanism

We exhaustively analyzed the MSI protocol and discovered that the design must be done carefully to garner a low WCL. We concentrate our discussion on two key design choices.

A. L2 cache controller design

In MSI [11], the L2 cache controller arbitrates requests between the forward, demand, and response networks in the following manner: the response has the highest priority followed by forward and finally demand. This priority order ensures that the causal order of messages is preserved [11]. Note that this prioritization exists in silicon-proven L2 designs [16]. However, this specific priority order results in a prohibitively large WCL. This occurs when cores request cache lines privately cached by a specific core, c_{ua}. This causes the directory to generate and send forward messages (asks c_{ua} to send data to requesting cores) to c_{ua}'s L2, which get queued in c_{ua}'s FRF. Due to forward messages having higher priority, c_{ua}'s demand request cannot be processed until all forward messages are processed. In the worst-case, a core's demand request is processed by L2 after all cache lines receive two forward requests because of this prioritization. This happens when all cache lines in L2 are in M state, where the first forward request requests the data be forwarded via a FwdGetS, and the second forward request asks c_{ua} to invalidate its copy via a Inv. In an 8kB L2 cache with 512 cache lines where the processing of each forward request takes 14 cycles, a core's request is delayed 14336 cycles in the worst-case before being processed by the L2.

Solution 1. The culprit causing the large WCL is the priority order of messages between the three networks. We observe that we can preserve the same properties necessary by the prioritization [11], but significantly lower the WCL by using a work-conserving round-robin (WCRR) between the demand and forward networks, while the response network remains the highest priority as illustrated in Figure 3a. With WCRR, a demand request waits for 14 cycles in the worst-case before it can be processed, and this delay is agnostic of the L2 size.

B. Coherence protocol: enabling predictable data sharing

MSI implementations require the directory to acknowledge a write-back request with a PutAck message [11]. This is necessary for correctness [11], [19]. Waiting for a PutAck results in a large WCL latency. We explain this with the example in Figure 3c with the contents of the DDRF on the right. Suppose core c_{ua} initially caches A exclusively in M state. c_{ua} issues a Store on B that requires it to evict A as a victim. This causes the PutM on A to be sent to the LLC to revoke its ownership at 1. This transitions c_{ua}'s copy of A to a transient state MI. MI indicates that requests for c_{ua}'s copy of A perceive the cache line in M, but after receiving the PutAck, it will be invalid. In parallel, c_2 issues a Write demand request to A at 2, and the directory forwards this to c_{ua} with a FwdGetM. At 3, c_3 issues a Write request to another cache line C, that arrives later than c_2's request. The directory processes these requests in arrival order due to the FIFOs. Hence, the directory issues FwdGetM to c_{ua}, where c_{ua} sends Data of A as response at 7. Since c_{ua} is still waiting for a PutAck, but it received the FwdGetM request, the cache line state transitions from MI to transient state II. This indicates that any further requests to c_{ua} for A will cause c_{ua} to respond with the copy not being valid. Note that by receiving the FwdGetM, c_{ua} knows that another

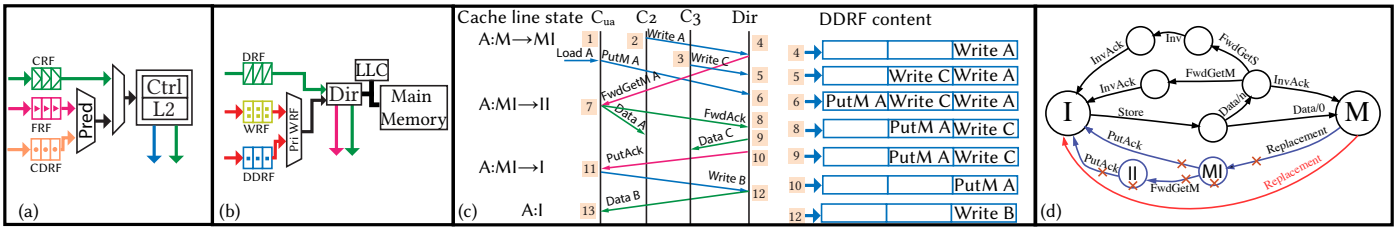

Fig. 3: (a) Ditty L2 cache has a controller (Ctrl) and storage and has an interface with predictable arbitration (Pred). (b) Directory (Dir) interface eliminates PutAck; WRF takes priority over DDRF. (c) PutAck introduces long latency. (d) Ditty eliminates PutAck.

core is writing to A; hence, it no longer has a valid copy of A. c_{ua} also sends a FwdAck to the directory at 7. The transition from M to I is highlighted in Figure 3d. Observe that when the directory receives c_{ua}'s PutM request, the requests by c_2 and c_3 have arrived at the directory before it; thus, delaying c_{ua}'s PutM as shown in the DDRF at 6. Hence, the directory must complete c_2's request for A, and c_3's request for C before processing the PutM from c_{ua} with a response of PutAck at 10. This allows c_{ua} to issue its next request to B. In the worst-case, all other cores, like c_2 and c_3, issue requests that arrive at the directory before c_{ua}'s PutM, forcing the WCL to assume $(N-1)$ requests to be completed before completing c_{ua}'s write-back request. This leads to a large WCL.

Solution 2. Notice that a write-back request is always a result of a demand request. Hence, the demand request must wait for the write-back request to complete before the L2 can perform the demand request. The PutAck signifies the end of the write-back request. Based on this observation, we remove the need to wait for a PutAck on a write-back request by combining the demand with the write-back request, and sending them together to the directory. This combination also prevents forward requests from intervening the process of the write-back and demand request. This freedom of interference from forward requests lowers the WCL. Ditty augments the coherence protocol and micro-architecture to implement this combining of requests. Figure 3d highlights coherence protocol changes for M to I transition, in which a replacement on M directly transitions the line into I without waiting for PutAck.

A naive implementation, such as using a single FIFO for demand and write-back requests, combined with the removal of PutAck can block requests indefinitely. Suppose the directory encounters a demand request to a cache line A, and A is written back by its owner with a PutM that queues after the demand request to A in the FIFO. The demand request is stalled since the directory cannot process the PutM request needed for the demand request to complete.

To prevent this situation, Ditty makes micro-architectural modifications as shown in Figure 3b. Specifically, Ditty uses two FIFOs, directory demand request FIFO (DDRF) and write-back request FIFO (WRF), to buffer demand and write-back requests, respectively. Then, the directory controller prioritizes the processing of write-back requests over demand requests. Note that responses continue to have the highest priority over both write-back and demand requests.

V. WCL Analysis for Memory Requests

We analyze the WCL a request experiences *after* the request misses in L1 and accesses the L2. Figure 4 illustrates the path the request takes in the hardware to experience the WCL. We start with a miss in the L1 resulting in a demand request being sent to the CDRF at the L2 (1). L2 sends this demand request over the request network to the directory (2). This request gets buffered in the DDRF and WRF of the directory (3). When the directory processes this request, it may invalidate copies of the cache line from cores by using the forward and response networks, and fetch the data from the main memory (4). Finally, the directory sends the data to the L2 (5).

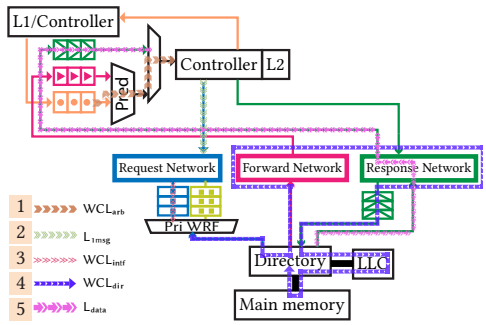

Fig. 4: A memory request of c_{ua} undergoes the WCL.

We begin by presenting the WCL at the directory (4 in Figure 4), WCL_{Dir} using Lemmas V.1 and V.2.

Lemma V.1. *The WCL a memory request experiences at the directory, WCL_{Dir}, is a demand request that is a miss in the directory and causes an eviction of a cache line from the LLC.*

Proof. Note that only write-back and demand requests arrive at the directory. Write-back requests access the directory and LLC's data array. Demand requests that miss in the directory require additional latency to access data from the main memory and send it through the interconnect to the requesting L2 in addition to having to write back a cache line. Thus, demand requests that miss in the directory experience the WCL. □

Lemma V.2. *The per-request WCL at the directory is:*

$$WCL_{\mathsf{Dir}} = \max\Big(\max(L_{\mathsf{Nmsg}} + L_{\mathsf{1msg}} + (2N-2)L_{\mathsf{L2}}, L_{\mathsf{mem}}),$$
$$L_{\mathsf{mem}} + \max(L_{\mathsf{1msg}} + L_{\mathsf{data}} + (2N-2)L_{\mathsf{L2}}, L_{\mathsf{mem}})\Big) + L_{\mathsf{Dir}},$$

where L_{1msg} is the WCL to transfer a non-data message over the predictable interconnect, L_{Nmsg} is the WCL to transfer N messages, L_{mem} is the WCL to transfer a cache line between the main memory and directory, L_{data} is the WCL to transfer a cache line between the LLC and L2, L_{L2} is the WCL of L2 processing one incoming message, and L_{Dir} denotes the WCL counterpart for the directory to access the internal structures.

Proof. Lemma V.1 states that the worst-case scenario occurs when c_{ua}'s request is a miss in the directory and causes an eviction in the LLC. The cache line selected for eviction, the victim, can be in one of three states: invalid (not cached in any private cache), shared or modified. We omit the case for an invalid victim as its eviction incurs no interconnect traffic.
Case 1. Victim is shared. A victim line that is shared, in the worst-case, may be privately cached by all N cores. Thus, writing back the victim requires invalidating all N copies due to inclusive cache hierarchy. The directory sends Inv messages to N cores and receives acknowledgments incurring a latency of L_{Nmsg}. When an Inv arrives at a core's L2, it may be last in its FRF. At worst, this Inv message is queued after $(N-1)$ forward requests in FRF (one less than N since there must be space in FRF for the Inv forward request itself). Additionally, this Inv forward request is a result of another core making a demand request, and given that a core can only make one outstanding demand request, we exclude the core making the demand request. Hence, the Inv message is actually queued after $(N-2)$ forward requests in the worst-case. With Solution 1, WCRR ensures the Inv is processed after $(N-1)$ demand requests and $(N-2)$ forward requests. Notice that a demand request can hit in the L2, whose completion allows a new demand request to appear in the next arbitration cycle. The demand requests, forward requests, and Inv itself each take L_{L2}, incurring a latency of $(2N-2)L_{\mathsf{L2}}$. Finally, it takes L_{1msg} to send back invalidation acknowledgement. When evicting the victim, the directory requests the main memory to fetch the data in parallel, incurring a latency of L_{mem}. This overlapping is possible because the invalidation only occupies the bandwidth of the interconnect and not the bandwidth of the main memory. Hence, the WCL is $\max(L_{\mathsf{Nmsg}}+L_{\mathsf{1msg}}+(2N-2)L_{\mathsf{L2}}, L_{\mathsf{mem}})$.
Case 2. Victim is modified. When the victim is modified by a core, the directory sends a FwdGetM to the core caching the data, causing a latency of L_{1msg}. Similar to the invalidation case, a FwdGetM is processed after $(N-1)$ demand requests and after $(N-2)$ forward requests in the worst-case, causing a total latency of $(2N-2)L_{\mathsf{L2}}$. The core then sends the data back to the directory, causing L_{data}. Finally, the directory needs to write-back the data to free an entry in the LLC, causing a latency of L_{mem}. The directory fetches the data in parallel, causing a latency of L_{mem}. Hence, the worst-case latency is $\max(L_{\mathsf{1msg}}+L_{\mathsf{data}}+(2N-2)L_{\mathsf{L2}}, L_{\mathsf{mem}})+L_{\mathsf{mem}}$. □

Lemma V.3 shows the critical instance of a memory request the core under analysis, c_{ua}, experiences.

Lemma V.3. *c_{ua}'s request experiences the WCL when (1) the request is processed after a forward request in the L2 cache; (2) the request triggers a replacement of a dirty cache line, incurring a write-back request in the L2; (3) the combined demand request and write-back request undergoes WCCL to reach the directory; (4) the request arrives at the DDRF of the directory and is the last of N demand requests in the DDRF and N write-back requests in the WRF; (5) before the request is processed by the LLC, the directory processes $(2N-1)$ write-back requests; (6) all N demand requests experience WCL_{Dir}, and, (7) the response undergoes the WCCL to reach the L2.*

Proof. We employ Figure 4 to guide the proof. (1) A demand request arriving at the L2 in CDRF is assured to be the only request since each core can only have one outstanding memory request. Thus, in the worst-case, this request is processed after one forward request due to the WCRR arbitration (1). (2) When L2 processes this request, the request happens to be a miss. This requires a victim line in L2 to be replaced. However, consider this victim line to be modified. Then, a write-back request (PutM) is necessary to vacate this entry. Thus, the L2 cache controller combines the demand and write-back request, and sends it to the directory. (3) This combined request takes WCCL to arrive at the directory (2). (4) At the directory, write-back requests have higher priority than demand requests; hence, in the worst-case, the PutM of the combined request interferes with its demand request. Further, the directory processes incoming requests from DDRF and WRF one at a time. In the worst-case, the demand request is last in the DDRF. Since each core has at most one outstanding request, all prior $(N-1)$ demand requests in DDRF have their corresponding write-back requests. Thus, in the worst-case, there are N write-back requests when the combined request arrives at the directory. (5) Before the directory processes the combined request, $(N-1)$ cores can finish their requests (taking WCL_{Dir} each) that arrived earlier in the DDRF, and they can issue new combined requests. Although the new demand requests are queued later in DDRF, their write-back requests interfere with demand request because the directory prioritizes the write-back requests. At worst, the interference is caused by $2N-1$ write-back requests. (4), (5) and (6) corresponds to 3 and 4 in Figure 4. (7) Finally, the directory sends back the data response incurring WCCL on the response network (5). □

Theorem V.4 sums the WCL of a memory request.

Theorem V.4. *The WCL of c_{ua}'s memory request is*

$$WCL = 2(L_{\mathsf{L2}} + L_{\mathsf{data}}) + WCL_{inter} + WCL_{\mathsf{Dir}},$$

where $WCL_{inter} = (2N-1)L_{\mathsf{Dir}} + (N-1)WCL_{\mathsf{Dir}}$ is the interference caused by demand requests and write-back requests at the DDRF and WRF.

VI. Evaluation

We evaluate Ditty in hardware on a setup with 2, 4, 6, 8, and 10 cores respectively, with a cache line size of 16-byte on the Amazon F1 FPGA using Xilinx Vitis 2022.1. We assume a 2-way set-associative L2 cache with a capacity of 512 B, and a 4-way set-associative inclusive LLC with 32 kB capacity. Our hardware implementation operates at 100 MHz, and reveals

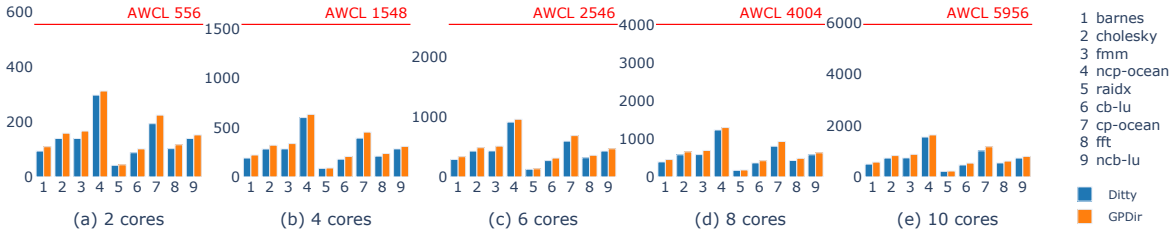

Fig. 5: Observed WCL in cycles.

that $L_{\text{L2}} = 14$ cycles, $L_{\text{Dir}} = 30$ cycles, and $L_{\text{mem}} = 200$ cycles. The left part of Table II shows the interconnect latency components corresponding to each of the configurations.

A. Synthetic workloads

The synthetic workloads stress our implementation of Ditty by accessing cache lines in the same set in the LLC. We also compare against a general purpose coherent data sharing mechanism (GPDir). Table II shows the observed WCL (OWCL) are below the analytical WCL (AWCL) for all configurations.

B. SPLASH-3 benchmarks

We evaluate Ditty and GPDir with SPLASH-3 benchmarks. Figure 5 shows the OWCL bound for Ditty and GPDir. We observe that as the number of cores increases, the WCL exhibited by Ditty is within the AWCL bound. Although our experiments show that GPDir has a similar OWCL, its WCL is not guaranteed as it is for Ditty. Our evaluation indicates that Ditty exhibits $1.12\times$ speedup compared to GPDir. Although we are able to execute `radiosity` and `raytrace`, due to the large input size, we are unable to finish the execution.

C. Resource Utilization

Table II shows the resource utilization of Ditty and GPDir private caches. Across all configurations, we observe that each Ditty private L2 cache consumes more flip-flops (FFs) and lookup tables (LUTs), and similar block RAMs (BR) compared to GPDir private L2 cache. This is because Ditty combines the demand request and its corresponding write-back request, hence, the vacant entry is available for tracking the demand request.

Ditty's LLC consumes 5795 FFs, 9997 LUTs, 116 BRAMs and 8 URAMs. While the GPDir consumes 5876 FFs, 10153 LUTs, 116 BRAMs and 8 URAMs. Hence, Ditty consumes similar FFs compared to a GPDir. Hence, Ditty's use of DDRF and WRF to store demand requests and write-back requests does not cause significant overhead.

TABLE II: Interconnect latencies (Intc. Lat.), observed worst-case latency (OWL), analytical worst-case latency (AWL), and resource utilization of Ditty and GPDir.

	Intc. Lat. (cycles)		WCLs (cycles) Ditty		GPDir	Resource utilization Ditty			GPDir		
N	$L_{1\text{msg}}$	$L_{N\text{msg}}$	OWL	AWL	OWL	FFs	LUTs	BR	FFs	LUTs	BR
2	4	7	507	556	511	5503	8326	4	5710	9156	4
4	10	32	849	1548	859	10917	16588	8	11328	18248	8
6	19	74	1177	2546	1164	16393	24870	12	16992	27360	12
8	34	130	1498	4004	1510	21791	33144	16	22621	36464	16
10	51	204	1882	5956	1885	27261	41438	20	28326	45588	20

VII. Conclusions

Ditty is a predictable cache coherent data sharing mechanism that (1) predictably arbitrates between demand requests and forward requests at the L2, (2) prioritizes WRF at the directory, and (3) eliminates PutAck in the coherence protocol to achieve a lower WCL bound. Our empirical evaluation shows that Ditty imposes negligible impact on performance compared to its general purpose counterpart.

References

[1] A. M. Kaushik *et al.*, "A Systematic Approach to Achieving Tight Worst-Case Latency and High-Performance Under Predictable Cache Coherence," in *RTAS*, 2021, pp. 105–117.

[2] S. Hessien *et al.*, "PISCOT: A Pipelined Split-Transaction COTS-Coherent Bus for Multi-Core Real-Time Systems," *ACM TECS*, July 2022.

[3] J. P. Cerrolaza *et al.*, "Multi-Core Devices for Safety-Critical Systems: A Survey," *ACM Comput. Surv.*, vol. 53, no. 4, Aug. 2020.

[4] A. M. Kaushik *et al.*, "Designing Predictable Cache Coherence Protocols for Multi-Core Real-Time Systems," *IEEE Transactions on Computers*, vol. 70, no. 12, pp. 2098–2111, 2020.

[5] R. Mirosanlou *et al.*, "Parallelism-Aware High-Performance Cache Coherence with Tight Latency Bounds," in *ECRTS*, vol. 231, Dagstuhl, Germany, 2022, pp. 16:1–16:27.

[6] R. Pujol *et al.*, "Empirical Evidence for MPSoCs in Critical Systems: The Case of NXP's T2080 Cache Coherence," in *DATE*, 2021, pp. 1162–1165.

[7] R. Kumar *et al.*, "Interconnections in multi-core architectures: Understanding mechanisms, overheads and scaling," in *ISCA*, 2005, pp. 408–419.

[8] B. de Dinechin, "Consolidating High-Integrity, High-Performance, and Cyber-Security Functions on a Manycore," in *DAC*, 2019, pp. 1–4.

[9] D. Chaiken *et al.*, "Directory-based cache coherence in large-scale multiprocessors," *Computer*, vol. 23, no. 6, pp. 49–58, 1990.

[10] S. Wasly *et al.*, "HopliteRT: An efficient FPGA NoC for real-time applications," in *ICFPT*, 2017, pp. 64–71.

[11] V. Nagarajan *et al.*, "A Primer on Memory Consistency and Cache Coherence, Second Edition," vol. 15, no. 1, pp. 1–294, Feb. 2020.

[12] S. M. Tam *et al.*, "SkyLake-SP: A 14nm 28-Core xeon® processor," in *ISSCC*, 2018, pp. 34–36.

[13] P. Conway *et al.*, "Cache hierarchy and memory subsystem of the amd opteron processor," *IEEE Micro*, vol. 30, no. 2, pp. 16–29, 2010.

[14] M. Hossam *et al.*, "Predictably and Efficiently Integrating COTS Cache Coherence in Real-Time Systems," in *ECRTS*, 2022, pp. 17:1–17:23.

[15] F. Rehm *et al.*, "The Road towards Predictable Automotive High - Performance Platforms," in *DATE*, 2021, pp. 1915–1924.

[16] J. Balkind *et al.*, "OpenPiton: An open source manycore research framework," in *ASPLOS*, 2016, p. 217–232.

[17] Z. Shi *et al.*, "Real-Time Communication Analysis for On-Chip Networks with Wormhole Switching," in *NOCS*, 2008, p. 161–170.

[18] J. Zuckerman *et al.*, "Enabling Heterogeneous Multicore SoC Research with RISC-V and ESP," in *CARRV*, 2022.

[19] R. Komuravelli *et al.*, "Revisiting the Complexity of Hardware Cache Coherence and Some Implications," *ACM TACO*, vol. 11, no. 4, dec 2014.