Addressing Resource Contention and Timing Predictability for Multi-Core Architectures with Shared Memory Interconnects

Haitong Wang, Neil C. Audsley, Wanli Chang
Department of Computer Science, University of York, UK
{hw963,neil.audsley,wanli.chang}@york.ac.uk

Abstract—**Multi-core architectures are increasingly being used in real-time embedded systems. In general, such systems have more processors than the shared memory modules, potentially causing severe interference over memory accesses. This resource contention could lead to substantial variation on memory access latencies, and thus wide fluctuation in the overall system performance, which is highly undesirable especially for the time-critical applications. In this paper, we address resource contention and timing predictability for multi-core architectures with distributed memory interconnects. We focus on the *locally arbitrated* interconnect constructed by pipelined multiplexing stages with local arbitration, while the *globally arbitrated* interconnect employing global scheduling to the same architecture potentially suffers synchronisation issue and requires strict coordination. Our contributions are mainly threefold: (i) We analyse the resource contention across the memory access data path, and report the accurate calculational method to bound the worst-case behaviour. (ii) We compare the average-case behaviour of the *locally arbitrated* and the *globally arbitrated* architectures with experiments, demonstrating varying memory latencies caused by the resource sharing issue. (iii) We propose an architectural modification to smooth resource sharing. Evaluations on simulators and FPGA implementations with synthetic memory workload show that the latency variation is significantly reduced, contributing towards timing predictability of multi-core systems.**

I. Introduction

As a growing number of applications with complex functionalities are integrated into the modern real-time embedded systems, such as in the emerging domains of autonomous vehicles and robotics, multi-core and network-on-chip (NoC) [1] [2] architectures are increasingly being used to achieve high performance. In general, there are more processors than shared memory modules in these systems. This potentially causes contention over memory accesses, which will get more severe with the trend of integrating more processors. Such contention could lead to substantial varying memory latency, and thus wide fluctuation in the overall system performance, which harms time-critical applications.

As illustrated in Figure 1, the processor stalls with different slack time, depending on the varying memory response time. Therefore, the memory access latency variation directly influences the processor utilisation and the dependent processes. In addition, such latency variation leads to very pessimistic worst-case assumptions in the timing analysis — where the maximum contention has to be assumed for most, if not all, memory accesses — and hence large safety margins.

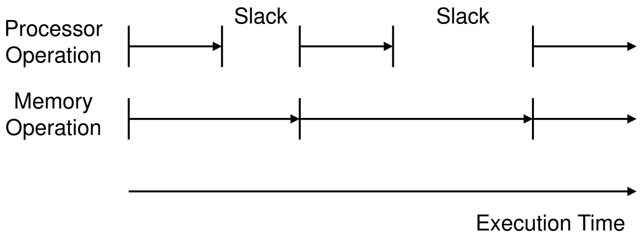

Fig. 1: Processor Operation versus Memory Operation

The current multi-core architectures trend to employ distributed memory interconnects rather than the conventional centralised interconnect. The distributed design deploys a tree-based structure with pipelined stages to break the critical path of the multiplexing into multiple smaller steps with smaller logic size. More detailed explanation can be found a bit later in Section III as illustrated by Figure 2. Although this design introduces additional delays in terms of clock cycles, the distributed data path allows higher synthesisable clock frequency. It scales to a large number of processors. The *locally arbitrated* interconnect is simply constructed by pipelined multiplexing stages with local arbitration schemes. It allows the average-case latency to be much lower than the worst case. However, the latency variation is large. Alternatively, the *globally arbitrated* interconnect employs global scheduling schemes to the same architecture, based on the pipelined data path. It budgets each processor with a limited memory bandwidth partition. This reduces the contention to the shared resources. However, the reservation can waste the system bandwidth and slow down the processors. The average-case performance is then degraded. In addition, the design also requires strict coordination and potentially suffers the synchronisation issue.

Main contributions: In this paper, we aim to address the resource contention and the timing predictability for the multi-core architectures with the distributed memory interconnects. Firstly, we analyse the hardware resource contention across the memory access data path, with the focus on the *locally arbitrated* platform. We define the general flow of the predictability analysis considering the blocking effect caused by the critical resource contention, with the accurate calculational method to bound the worst case. Secondly, we compare the average-case behaviour of the *locally arbitrated* and the *globally arbitrated* architectures using experiments, with the analysis of the memory latency variation caused by the resource sharing issue. Thirdly, we present an effective architectural modification to the *locally arbitrated* multi-core architecture to smooth the resource sharing. It is to employ an additional hardware queue between the interconnect root and the shared memory module. Experiments on simulators and FPGA implementations show that the latency variation is significantly reduced, contributing towards the timing predictability across the studied platform.

The remainder of this paper is structured as follows. Section II reviews the related work in multi-core memory interconnects and critical resource contention. Section III analyses the resource contention across the *locally arbitrated* shared-memory multi-core architectures with predictable behaviour, and proposes the accurate calculational method to bound the worst case. Section IV presents the comparison between the *locally arbitrated* and *globally arbitrated* interconnects, with analysis on the resource fairness issue and memory latency variation. The root queue modification to smooth the recourse sharing across the *locally arbitrated* platform is introduced in Section V. Section VI gives the evaluation, including hardware simulations and FPGA experiments. The related analysis follows up. Section VII draws the conclusion.

II. Related Work

This section presents the literature review, including the multi-core memory interconnects and the critical resource contention within such architectures.

A. Memory Interconnects

The conventional multi-core architecture employs a shared bus to connect processors and the shared memory (e.g. the AHB bus [3] in the SoC design). Communications between processors or accesses between processors and the memory must be delivered through the shared bus. Once a single access occurs, the bus is blocked. This leads to serious contention. Alternatively, the crossbar interconnect alleviates the contention issue with a set of switch boxes, using dedicated links to replace the shared bus, such as the AXI interconnect [4]. This allows multiple accesses to occur simultaneously. The NoC architecture employs a packet switching network [5] [6]. Each processor connects through a router to the communication network. In this way, a processor can access its target with less bus contention. Commonly, the shared memory is connected to the edge of the router network.

With the aim to predict the behaviour of memory access, the architectures above typically utilise an arbitration scheme (e.g. priority-based, TDM or round-robin) to provide timing guarantees. The conventional centralised implementation of arbitration schemes employs a single arbiter, allowing arbitration decisions to be made at the central location. However, as the number of processors increases, the logic size of the arbiter increases. This limits the maximum synthesisable clock frequency. One promising approach is to use distributed memory interconnects. It deploys the tree-based structure with pipelined stages to break the critical path of the multiplexing into multiple smaller steps with small logic size. Although this introduces additional delays in terms of clock cycles, the distributed interconnect allows higher synthesisable clock frequency, and scales to a large number of processors.

The *locally arbitrated* distributed interconnect is constructed based on a binary arbitration tree that multiplexes the memory requests from the processors to the shared memory module. For example, [7] develops the arbitration tree with the globally synchronised timestamps. Then the arbitration at each local distributed multiplexing stage operates the first-come-first-served (FCFS) scheme that the memory request with a lower timestamp will be allowed to relay. However, the application is only feasible to few platforms, such as the system employing AXI bus [8] with very limited number of outstanding memory requests.

Alternatively, Bluetree [9] [10] is initially developed for the NoC architecture. It is the external memory tree which provides a second network exclusive for the accesses or communications to the shared memory. This separates the memory traffic from the processor router network. In this way, memory accesses no longer interfere with the communications between the processors. Bluetree interconnect is constructed by a set of pipelined multiplexers using local round-robin arbitration scheme, and the Bluetree memory architecture does not require full synchronisation. Besides that, it also allows multiple memory requests to be in the transfer through the tree network simultaneously. This aids further scalability. However, the *locally arbitrated* interconnect requires complicated analysis on the predictable behaviour.

By contrast, the *globally arbitrated* interconnect integrates the global scheduling scheme with the distributed multiplexing. For example, TDM Tree [11] is constructed by the global TDM scheduling components and the distributed tree network. When the TDM time slot arrives, one memory request from a processor is allowed to relay to the tree network. With the global scheduling interval, there is no contention to the shared resource, neither the tree data path nor the root memory module. However, the TDM tree requires strict synchronisation and complex coordination. In addition, it does not support work-conservation. This can waste bandwidth.

Based on the global scheduling interval, Globally Arbitrated Memory Tree (GAMT) [12] [13] extends the distributed tree with priority-based rate control, such as the Frame-based Static Priority (FBSP) and the Credit-Controlled Static Priority (CCSP). With the aim to utilise the bandwidth with flexibility, GAMT allows successive memory requests from one processor to relay at a time. It can benefit specific applications.

B. Critical Resource Contention

In this paper, we address the resource contention over multi-core architectures with distributed memory interconnects. Such architectures are typically designed for average-case performances, with inevitable interference from the software components or tasks. The consequent contention to the shared hardware resources may block the flow of memory requests and communication packets. It may also block any subsequent flow, even causing the resource fairness issue. The contention to the critical resources, such as the memory module and hardware data path, potentially leads to varying latency.

The impact of the critical resource contention has been widely discussed within the multi-core or many-core architectures, especially in the NoC applications [14] [15]. The contention to the shared router blocks the flow of communication packets, leading to varying edge-to-edge latency across the processor router network. With the aim to regulate the access to a single shared router, [15] [16] introduce the wormhole switching with credit-based or priority-based flow rate control schemes. [17] presents the design of channel tree with reserved time slots to achieve contention-free routing in the network. The alternative method is to employ virtual channel [18] [19], which provides flexibility in the channel utilisation.

By contrast, the tree-based architecture appears more sensitive to the blocking caused by the contention to the critical resource. For example, the *locally arbitrated* architecture allows multiple memory requests in transfer simultaneously, leading to the contention to the shared root memory module. The entire interconnect network may also be affected by any blocking in the overlapped request paths, especially with the blocking closer to the tree root. When there is one request occupying the memory module, many others stall, just waiting in the shared interconnect paths. It blocks the entire tree network and the subsequent requests as well. With the *locally arbitrated* multiplexing stages, these pending requests may be further blocked by the newly issued requests. The sequence of the pending requests is broken, and the memory bandwidth is not fairly shared. Requests suffer additional latency, and the actual latency could vary substantially at runtime. Such latency variation requires complex timing analysis.

According to the previous analysis, the *globally arbitrated* interconnect integrates the global scheduling scheme with the distributed multiplexing. For example, due to the globally scheduled time slots, TDM Tree provides the contention-free request paths. Based on the global scheduling interval, GAMT employs additional rate control schemes. These architectures avoid the resource contention, and there is no resource sharing issue. However, those memory requests stalled due to the timing division may suffer additional latency, and the latency variation is related to the global timing interval. If the memory requests are distributed in time, they must wait for the strict scheduling cycle, increasing the latency proportional to the global cycle, and resulting in substantial variation.

The alternative solution to alleviate the critical resource contention within the tree-based architecture is message combining [20]. For the memory interconnect with multiple pipelined stages, the requests simultaneously arriving at one arbiter stage can be merged. This reduces the contention to both the request path and the shared root memory module. The memory response is then split to multiple individual ones during the response path. This method leaves the design burden to the root memory controller. Besides, it requires increasing logic size for each pipelined stage in the data path, limiting the synthesisable clock frequency.

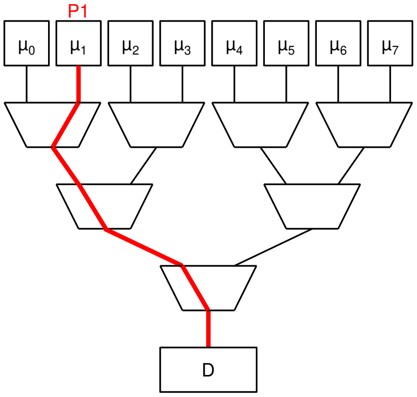

Fig. 2: 8-client Bluetree Architecture

III. Predictable Resource Contention Behaviour across the Locally Arbitrated Architecture

Our work starts with the resource contention analysis over the multi-core architectures with distributed memory interconnects. We focus on the predictable behaviour analysis of the *locally arbitrated* Bluetree which provides good average-case performance and guarantees the worst-case latency. The Bluetree-based multi-core application is shown in [21], which shows promising performance, although the behaviour analysis is very limited due to the architectural choice [22]. In this section, we provide the predictable behaviour analysis considering the resource contention. It involves the analysis of the blocking effects within the Bluetree memory architecture, and gives the accurate calculational worst-case bound.

A. Bluetree Architecture

Figure 2 shows the 8-client Bluetree memory architecture. It consists of the clients, the Bluetree interconnect, and the shared memory module. A client can be a single processing core or a multi-core processor. It is marked as μ_j, where j is for the client index. Each client also has its individual memory access path P_j, such as path P_1 for client μ_1 as highlighted in the graph. The Bluetree interconnect B employs multiple stages of 2-to-1 multiplexers to construct the tree network, connecting multiple clients at the tree leaves to the shared memory module D at the tree root. When a client issues a memory request, this request is multiplexed and relayed to the shared memory across the Bluetree network. Then the memory response returns to the corresponding client across the bi-directional Bluetree network. As the number of client increases, the tree network scales with more Bluetree multiplexer stages. This increases the Bluetree depth N_β.

The design of the Bluetree multiplexer is shown in Figure 3. Arbitration occurs in the request path (RQ) to decide which request from either the client direction to be relayed to the memory direction, potentially to the next Bluetree multiplexers. The blocking factor α of the internal arbiter is defined such that every α requests from *path 0* can be

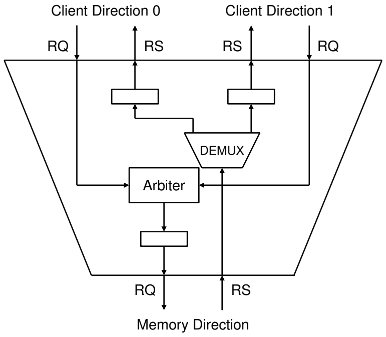

Fig. 3: Bluetree Multiplexer

blocked by at most one request from *path 1*. In this way, *path 0* can be considered as the local high-priority path, and *path 1* is the local low-priority, with the caveat that starvation can be prevented by allowing one request from the local low-priority path to be eventually relayed. If there is no request from *path 0*, the arbiter allows no blocking to *path 1* with an outstanding request. Therefore, every single request from the local low-priority path can be blocked by up to α requests from the local high-priority path. In contrast, the response path (RS) is non-blocking (in any Bluetree multiplexor). The internal demultiplexer simply decides the route direction of the memory response. Besides that, a buffer is employed in each local path as the common pipeline design.

The Bluetree memory architecture is designed to provide good average-case performance and guarantee the worst-case memory latency. However, the shared root memory is the architectural bottleneck. As shown in Figure 2, closer to the Bluetree root, more memory access paths overlap. Therefore, the memory requests from different clients have to share the common hardware paths, as well as the shared root memory. This shared interconnect architecture inevitably introduces the resource contention over simultaneous memory accesses.

The behaviour analysis focuses on the memory latency across the Bluetree architecture. It involves the integration of the shared root memory into the system. The latency t of a memory request ω consists of request path latency t_{RQ}, root memory latency t_D, and response path latency t_{RS} as follows:

$$t(\omega) = t_{RQ}(\omega) + t_D + t_{RS}(\omega) \tag{1}$$

In this paper, the root memory latency t_D is considered as a fixed constant to simplify the analysis. The response path latency t_{RS} defines the latency across the Bluetree network B from the memory to the client. With the pipelined buffers in the response path, it requires 1 clock cycle to cross one Bluetree stage. The analysis of the basic request path latency t_{RQ} is similar. However, when there is contention in the request path, the memory request ω can be blocked by other requests. If the request ω is blocked, the corresponding request path latency t_{RQ} inevitably increases. This increases the total latency $t(\omega)$.

B. Worst-Case Behaviour

With the analysis above, blocking only occurs in the request path, the worst-case latency t^{WC} requires the determination of the worst-case latency in the request path t_{RQ}^{WC} as follows:

$$t^{WC}(\omega) = t_{RQ}^{WC} + t_D + t_{RS}(\omega) \tag{2}$$

For each blocking that the request ω suffers, the maximum overall latency will increase by an amount proportional to the root memory latency t_D - essentially the request flow actually stalls until the shared memory is empty again to accept a next request. The root memory latency can mask the buffer latency across the Bluetree path, as requests can proceed within the Bluetree until blocked in parallel to the memory responding to the current request. Therefore, to determine the worst-case latency in the request path t_{RQ}^{WC} requires to determine the related maximum blocking number N_{RQ}^{WC}, and the relationship between t_{RQ}^{WC} and N_{RQ}^{WC} is defined as follows:

$$t_{RQ}^{WC} = N_{RQ}^{WC} \times t_D \tag{3}$$

Therefore, the worst-case memory latency with Bluetree depth N_β can be refined from equation 2 as follows:

$$t^{WC}(\omega) = (N_{RQ}^{WC} + 1) \times t_D + N_\beta \tag{4}$$

In [22], the maximum blocking number is determined by a simulation-based method for a specific Bluetree application, which utilises the AXI bus [8] between Bluetree multiplexers and the shared memory module. It only considers very limited blocking (i.e. blocking effect is limited by architectural choice). In this section, we provide the accurate blocking analysis considering all blocking effects. The maximum blocking number will be determined with calculational method.

Blocking in the Bluetree memory architecture can be classified as *inter-path blocking* and *intra-path blocking*. *Inter-path blocking* is when request ω transfers across a Bluetree multiplexer and is blocked by some other request from the other local path. As shown in the previous analysis, *inter-path blocking* is affected by the local blocking factor α. By contrast, *intra-path blocking* is when one memory request ω is blocked by any other request ahead of it, either from the same client or from other clients. According to the nature of the architecture, there are more *intra-path blocking* closer to the Bluetree root. The interaction of *inter-path blocking* and *intra-path blocking* also needs to be included. When request ω is interfered by a single *inter-path blocking*, one request from a different path overtakes ω, becoming the request ahead in their overlapping path. This can lead to additional *intra-path blocking*.

As the Bluetree depth N_β increases, the blocking analysis complicates. First of all, the number of Bluetree buffers in one request path increases. This increases the *intra-path blocking*. Secondly, *inter-path blocking* increases as the number of Bluetree arbiters increases. Also, there is interference between different Bluetree stages. According to the nature of the tree-based architecture, if there is any blocking in the stage closer to the root, the entire Bluetree interconnect will be affected. For example, if the Bluetree root stage β_0 is blocked, the request flow in all Bluetree paths stall, with the Bluetree blocking counters stalled but not updated. Therefore, if there are more *inter-path blocking* in the level closer to the clients at the tree leaf level, there will be more consequent *intra-path blocking* in the overlapping paths. The interference between Bluetree stages becomes serious, and the maximum blocking number N_{RQ}^{WC} increases significantly.

Based on the analysis above, *priority path* is introduced to analyse the maximum blocking number in the request path N_{RQ}^{WC}. It is to track the local priority at each Bluetree stage of one request path, from the client to the shared memory. For example, as shown in Figure 2, *priority path* P_1 for client μ_1 is $P_1 = \{L, H, H\}$, where L is for local low-priority and H for local high-priority. Then path P_1 is across the local low-priority path at Bluetree stage β_2, the local high-priority path at β_1, and the local high-priority path at the Bluetree root stage β_0 to the memory. The related local priority expressions are $P_1(\beta_2) = L$, $P_1(\beta_1) = H$, and $P_1(\beta_0) = H$.

The calculation of N_{RQ}^{WC} for one complete request path is iterative, based on the calculation of the maximum blocking number at each separate Bluetree stage. The intuition behind this method is that, the blocking number at any given Bluetree stage is dependent on (1) the amount of blocking has occurred at previous stages along the request path, and (2) the amount of blocking can occur at the current stage (which is dependent on α). $N_{RQ}^{WC}(\beta_i)$ is defined as the iterative blocking upto and including the stage β_i, and maximum arbiter blocking number $N_{\alpha}^{WC}(\beta_i)$ is to represent the blocking at stage β_i only. Therefore, the iterative calculation can be expressed as follows, where *plus 1* represents the local buffer is occupied:

$$N_{RQ}^{WC}(\beta_i) = N_{RQ}^{WC}(\beta_{i+1}) + N_{\alpha}^{WC}(\beta_i) + 1 \tag{5}$$

The maximum arbiter blocking number $N_{\alpha}^{WC}(\beta_i)$ is decided by the local blocking factor α at the Bluetree stage. According to the previous analysis, every α requests from the local high-priority path can be blocked by at most one request from the local low-priority path, and every single request from the local low-priority path can be blocked by upto α requests from the local high-priority path. Therefore, the maximum arbiter blocking number $N_{\alpha}^{WC}(\beta_i)$ at one Bluetree stage can be calculated as follows:

$$N_{\alpha}^{WC}(\beta_i) = \begin{cases} \lceil \frac{(N_{RQ}^{WC}(\beta_{i+1})+1)}{\alpha} \rceil & \text{H} \\ (N_{RQ}^{WC}(\beta_{i+1}) + 1) \times \alpha & \text{L} \end{cases} \tag{6}$$

In summary, the maximum blocking number in the request path N_{RQ}^{WC} can be determined with the iterative calculation stages from the client to the Bluetree root, with assumptions that (1) all the pipelined buffers are occupied, and (2) each local Bluetree arbiter always harms the request flow. Request ω in one *priority path* gives $\omega \in P_j$. Equation (5) and (6) bound the maximum blocking number at each Bluetree stage $N_{RQ}^{WC}(\beta_i)$ with local priority $P_j(\beta_i)$. In this way, the maximum blocking number N_{RQ}^{WC} is calculated iteratively.

Obviously, with the blocking factor α increasing, the maximum blocking number N_{RQ}^{WC} decreases in the request path with more local high-priority tracks, while N_{RQ}^{WC} increases with more local low-priority tracks. The calculation can also be extended that different blocking factor α values can be set at each Bluetree stage. By contrast, when the blocking factor is set as $\alpha = 1$, the Bluetree can be considered as the distributed binary tree stages with local round-robin scheme. This provides the relatively fair access to the shared memory module for all clients. It remains as default in later sections.

In this section, our method first defines the general analytical flow for the predictable behaviour of the *locally arbitrated* platform. It can be easily extended to multi-core architectures using a different *locally arbitrated* memory interconnect, with modification to the local arbitration calculation.

IV. Locally Arbitrated vs. Globally Arbitrated

Section III presents the resource contention analysis in multi-core architectures with distributed memory interconnects. It also defines the general analysis flow of the *locally arbitrated* interconnect. The memory access over such architectures shows predictable behaviour. If there is uncertainty with respect to the memory request numbers or the memory request issuing time instants, the worst-case memory access latency across the multi-core architecture can also be bounded. However, this inevitably leads to pessimistic results. If the exact memory access profiles can be provided, the detailed blocking analysis requires accurate knowledge of the local arbiter states and memory request flow states. This requires increased complexity. Besides that, due to the variable blocking behaviour within the *locally arbitrated* architecture, the memory access latency can vary severely.

By contrast, the *globally arbitrated* interconnect integrates the global scheduling scheme with the distributed multiplexing stages. It can be considered as the *locally arbitrated* interconnect with path traffic shaping components. This real-time method aims to budget each processor with limited available memory bandwidth to achieve *temporal isolation*. It can reduce the hardware resource contention. However, sufficient reservations potentially waste the system bandwidth and slow down processors, degrading the overall system performance. For example, TDM Tree strictly shapes memory accesses to the shared resources and therefore eliminates the contention. However, memory requests can stall in TDM Tree even with empty interconnect and idle memory module simultaneously. GAMT employs additional rate control schemes based on the reserved time slots as compensation. It can benefit applications with successive memory requests. However, the *globally arbitrated* architecture suffers synchronisation issue. When memory requests are distributed in time, they must wait for the strict scheduling cycle, increasing the latency proportional to the global cycle, and resulting in substantial variation.

In this section, we compare behaviour difference between the *locally arbitrated* and the *globally arbitrated* interconnect, and unveil the latency variation within both architectures.

A. Average-Case Behaviour

The worst-case analysis above shows the behaviour when the system is flooded by memory requests. Due to the architectural features, the *locally arbitrated* interconnect allows multiple simultaneous requests, and this leads to contention to the shared hardware resource. The requests have to share the overlapped interconnect path as well as the root memory module. The contention increases latency. As more bandwidth is requested due to the increase of workload, more available system bandwidth is consumed. If the requested bandwidth keeps increasing, the system will saturate at some point, without delivering any additional bandwidth. Any further memory request will only have to wait for the service of the system. This saturation phenomenon commonly occurs with shared resource [23]. As shown in Section III-B, the saturation point of the Bluetree memory architecture can be determined by the worst-case analysis. It clearly bounds the maximum request number in one Bluetree path. Obviously, the workload pattern in the related worst-case assumption is independent of the response time. The client just keeps pushing requests into the system regardless of memory response.

In practical applications, the number of memory requests issued to the system will be limited, either by the characteristics of the application software, or by the architecture of a processor (e.g. maximum number of outstanding memory requests before the processor stalls). Besides that, the workload pattern is dependent on the memory response. The congestion still occurs due to the contention to the shared resource, and the latency increases. However, as the workload pattern is dependent on the response time, the client will slow down the request generation. Then the latency increase stops in turn. This dependency actually reflects the process of the practical applications. For example, a processor has to receive data from memory before any related operation. The characteristics of the workload pattern can be represented as follows:

$\boldsymbol{N_{RQ}(P_j)}$: The path outstanding request number, where j is for the path index. A client generates requests successively until the path limit. Then the client stalls, waiting for the response. Only when there is any response returned, another new request can be generated. The workload pattern is dependent on the response time. Besides that, the total system outstanding request number can be calculated with the sum operation.

$\boldsymbol{T_{RQ}(P_j)}$: The request interval between two successive memory requests. The client generates successive requests with intervals, normally in clock cycles. This distributes memory requests in time. It actually reflects the necessary processor execution time or the time across the data path in practical applications. Obviously, the appropriate amount of jitter can be introduced for the behaviour description. By contrast, when the interval is fixed as 1, the requests will be issued into the system more intensively. This keeps the corresponding path and the shared root memory module busy.

$N_{RQ}(P_j)$ and $T_{RQ}(P_j)$ can be combined to describe the path memory workload. This workload pattern is dependent on the response time, and it will be used to evaluate the multi-core interconnect in later sections. Obviously, the shared root memory module impacts the system performance. Any contention to this critical resource causes the congestion of the request flow and increases latency. As the path workload $N_{RQ}(P_j)$ increases or $T_{RQ}(P_j)$ decreases, the root memory is not able to response to the intensively incoming requests fast enough. The root module is actually in high demand but with limited bandwidth. In turn, memory access latency increases.

TABLE I: Path Outstanding Request Number

$N_{RQ}(P_j)$	0	1	2	3	4	5	6	7
a	0	0	0	2	1	0	0	0
b	1	0	1	2	2	0	0	1
c	2	1	1	3	3	1	1	1

B. Investigating Behaviour of Memory Interconnects

This section compares the average-case behaviour of the *locally arbitrated* Bluetree and the *globally arbitrated* TDM Tree. It is to evaluate the memory access latency of both architectures based on the 8-client system, with the assumption that both architectures are running with the same clock frequency.

1) Hardware Simulation: The initial experiments are performed by hardware simulations. We implement the system using Bluespec System Verilog (BSV) [24], with simulations running on BlueSim simulator. The shared memory module is implemented using BSV BRAM package [25] with extra delays as a constant $t_D = 20$ in clock cycles. A *traffic generator* is employed as a client instead of a processor. The *traffic generator* simulates memory requests without processing any data, and the workload pattern follows the analysis above.

In this experiment, each *traffic generator* issues 36 memory requests totally. The path request interval is fixed as $T_{RQ}(P_j) = 1$, and the path outstanding request number $N_{RQ}(P_j)$ varies as shown in Table I. The column is for path P_j, and the row is for three groups of memory workload combinations. The table content shows the increasing memory workload (from *group a* to *group c*) with the path outstanding request number increasing. The path *traffic generator* issues a memory request every clock cycle until the path limit $N_{RQ}(P_j)$. Then the *traffic generator* stalls. If there is any memory response returned, this *traffic generator* will issue a new memory request a next clock cycle.

With the setup above, the simulations run for Bluetree memory architecture and TDM Tree respectively. The measured metrics are release time and latency, both in clock cycles. The release time is to record the time point when the *traffic generator* issues one memory request refer to the global simulation time, and the latency is to record memory access time across each request path in the shared architecture. The experimental results are shown in Figure 4 and Figure 5 (please refer to the colourful version). The horizontal axis is for the release time, and the vertical axis is for the latency. The scatter plot is to show the latency variation.

Figure 4 (a) shows the performance of the 8-client Bluetree architecture with only memory requests in path P_3 and P_4. At the beginning of the simulation, the total system outstanding request number is $N_{RQ}(B) = 3$. As shown in the graph, the latency increases to approximately 60 very quickly. With the fixed request interval $T_{RQ}(P_3) = T_{RQ}(P_4) = 1$, the system shows the regular latency values. With different path outstanding request number $N_{RQ}(P_3) = 2$ and $N_{RQ}(P_4) = 1$ but with the same total request number, the path simulation

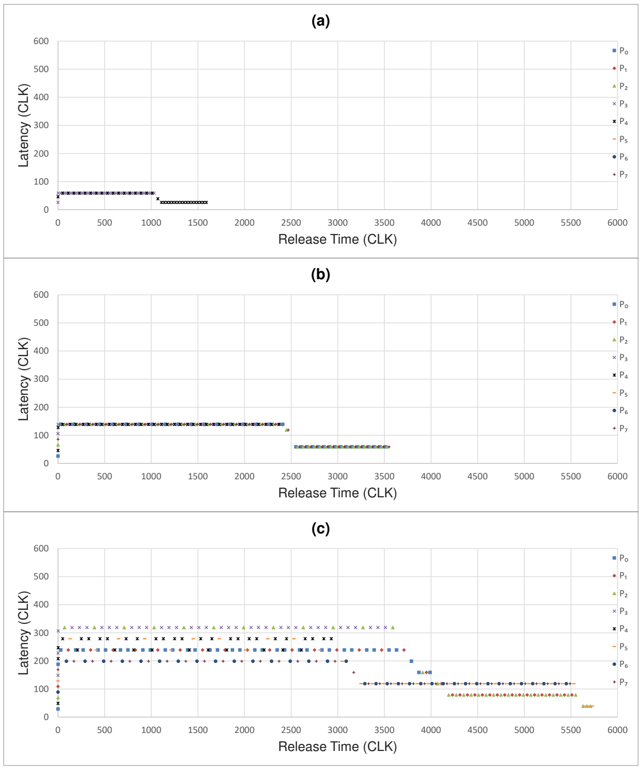

Fig. 4: 8-client Bluetree Performance

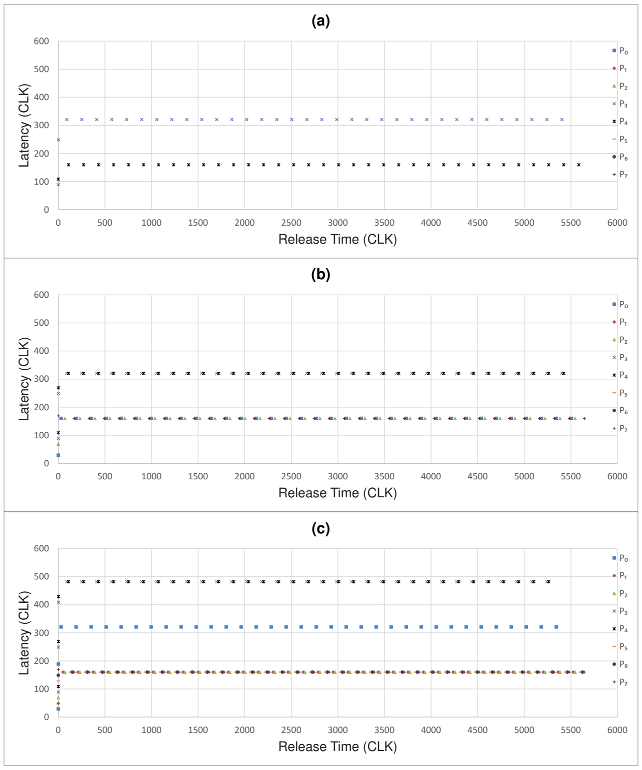

Fig. 5: 8-client TDM Tree Performance

completes at different time points. For example, the simulation in path P_3 with $N_{RQ}(P_3) = 2$ completes at approximately 1000. With the reduction of the system outstanding request number $N_{RQ}(B) = 1$, the contention to the memory reduces. The latency in path P_4 also decreases to approximately 20 until the end of the simulation. Figure 4 (b) shows the performance with the increased memory workloads. The latency variation shows the similar trend as in Figure 4 (a). By contrast, with the increased total system outstanding request number, the highest observed latency increases to approximately 150.

Figure 4 (c) shows the performance with further increased memory workloads. As shown in the graph, the latency for each path increases sharply in a very short period of time from the start period of the simulation. The Bluetree architecture actually becomes congested relatively quickly with intensively issued memory requests. Essentially, as the memory workload pattern is dependent on the response time, the rate of the request release drops. Then the latency increase stops in turn. In this way, the latency in each path tends to reach the corresponding maximum limit. Obviously, the fixed traffic parameters lead to these regular latency values. Besides that, the distribution of the scatters shows the latency variation. For example, the Bluetree latency in path P_4 and P_5 is approximately 280 or 240. With the contention to the shared resource, the latency varies due to the varying blocking behaviour. In the later period of the simulation, the latency decreases due to the reduction of of the system outstanding request number.

Figure 5 shows the performance of the 8-client TDM Tree architecture. Compared with Figure 4 (a), Figure 5 (a) clearly shows that TDM Tree does not support work conservation. With only memory requests in path P_3 and P_4, the interconnect or the memory module can be idle. However, the strict TDM only allows one memory request to be relayed to the empty data path at a time. As a result, the observed latency in path P_4 is approximately 160, and the simulation completes at approximately 5500. This reflects the global scheduling interval for 8 clients. Figure 5 (b) and Figure 5 (c) show the similar performance with increased memory workloads. By contrast, Bluetree employs local work-conserving round-robin scheme to provide good average-case performance.

In these experiments, the request interval is fixed as 1. A new memory request is issued immediately after the response returns. In this way, these memory requests can satisfy the TDM interval. As shown in Figure 5 (c), TDM Tree shows regular latencies which are easy to predict. For example, latency in path P_3 with $N_{RQ}(P_3) = 3$ is approximately 480. By contrast, Bluetree allows multiple memory requests in data path hence variable blocking behaviour as shown in Figure 4 (c). The inter-path interference also affects paths nearby. For example, P_5 with $N_{RQ}(P_5) = 1$ is severely affected by P_4 with $N_{RQ}(P_4) = 3$, and latency varies between 280 or 240.

2) FPGA Experiments: Further experiments are performed with FPGA implementation. We implement the 8-client system on Zedboard [26] (using Xilinx Vivado [27] [28]). The shared memory module is based on FPGA BRAM [29] with extra delays to fix as a constant $t_D = 20$. The *traffic generator* is employed as client. The synthetic memory workload includes path outstanding request number $N_{RQ}(P_j)$ and path request

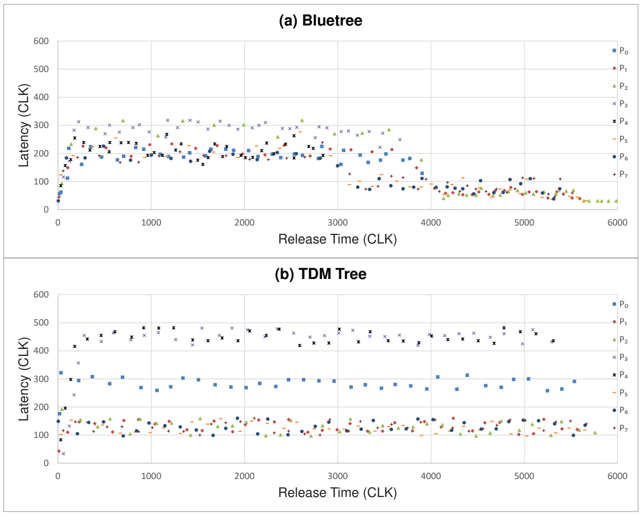

Fig. 6: 8-client Architecture Performance

interval $T_{RQ}(P_j)$. This provides varying memory workloads more close to the practical applications. The contents of the synthetic memory workload are stored in local BRAM for each client. In this experiment, the path outstanding request number $N_{RQ}(P_j)$ is set as *group c* in Table I, and the request interval $T_{RQ}(P_j)$ varies with randomly generated values between 1 to 64 as $T_{RQ}(P_j) \in [1, 64]$. The results are shown in Figure 6.

Figure 6 (a) shows Bluetree performance. With the introduction of the memory request interval, the latency varies following the similar trend as in Figure 4 (c) but with larger variation. With variable memory workloads, Bluetree shows variable behaviour and hence complex inter-path interference. For example, latency variation in P_4 increases from 40 to 100. By contrast, Figure 6 (b) shows TDM Tree performance. With variable request intervals, TDM Tree shows varying latencies. For example, latency variation in P_4 is approximately 100.

3) Discussion: This section shows the behaviour features of the *locally arbitrated* and the *globally arbitrated* interconnect. It also clearly demonstrates latency variation within the shared memory multi-core architecture. The *locally arbitrated* Bluetree provides good average-case performance. However, due to the variable blocking behaviour within Bluetree data path the timing predictability requires complex analysis on the accurate knowledge of the local arbiter states and the request flow states. By contrast, the *globally arbitrated* TDM Tree only provides regular latencies with memory workload satisfying strict timing interval. Besides that, TDM Tree potentially wastes bandwidth and Memory latency variation leads to wide fluctuations in overall system performance, and harms applications with real-time requirements.

V. Smoothing Resource Contention across Multi-Core Architectures

The multi-core architecture is typically designed for good average-case performance, and the resource contention within such architectures is inevitable. The contention to the critical resource, either the shared root memory or the overlapped data path, leads to varying memory access latencies. Section II-B discusses solutions to alleviate the resource sharing over the pipelined network. However, the tree architecture appears more sensitive to blocking caused by the resource contention.

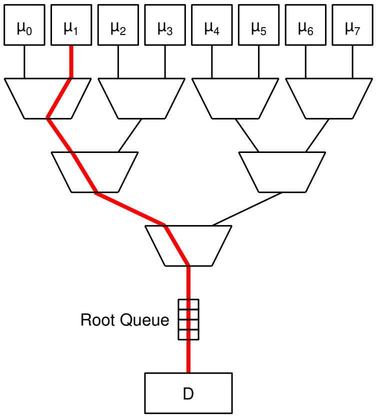

Fig. 7: Modified Bluetree Architecture with Root Queue

The key difference between the *locally arbitrated* interconnect and the *globally arbitrated* interconnect lies on the hardware resource sharing. The *locally arbitrated* interconnect employs local arbitration scheme along the distributed multiplexing data path, while the *globally arbitrated* interconnect isolates memory requests with global scheduling scheme. According to the analysis in Section IV-B, both architectures show varying memory access latencies. The *locally arbitrated* interconnect allows the average case to be much better than the worst case. However, the latency variation in the average case is inevitably substantial, and it requires complicated timing analysis. By contrast, the *globally arbitrated* interconnect allows the easy-to-predict behaviour with the assumption that the memory access profile strictly satisfies the global scheduling interval. It can benefit specific applications.

In this section, we present the architectural modification of multi-core platforms to smooth resource sharing and reduce memory latency variation. We improve the predictable memory interconnect to better support the real-time applications. This work is based on the *locally arbitrated* interconnect, and the Bluetree memory architecture is shown as an example. We modify the interconnect with an additional hardware queue. As shown in Figure 7, the queue is employed to connect the Bluetree root and the shared memory module. As request paths overlap to the root of the tree-based interconnect, each request will be relayed into the shared hardware queue. The root queue buffers the requests that arrive at the Bluetree root.

The design of the root queue is based on the bypass FIFO buffer. If the queue is empty, a request can be relayed to the memory directly without additional delays. If not empty, the queue temporarily stores the requests that arrive but cannot be immediately processed by the memory. The FIFO buffer also treats the queued requests equally, and the first-arrived request can be relayed to the memory first. This remains the arrival sequence of memory requests from the Bluetree interconnect, alleviating the contention over the overlapped paths.

With sufficient root queue size, all the outstanding memory requests can be stored in the buffers, rather than blocking the overlapped interconnect. In this way, there is no contention to the shared request paths. The root memory responses to these requests in FIFO sequence, and new arrival requests have to wait in queue behind. This can be defined as the *queued service* which smooths the resource sharing, and therefore reduces the latency variation across the architecture.

The premise of the *queued service* is that the size of the root queue is sufficiently large enough to store all outstanding memory requests in the system. Due to the architectural features, the *locally arbitrated* Bluetree interconnect also provides buffers as well as the root queue. The amount of the total queued buffers in this architecture is analysed as follows:

- The root memory provides 1 buffer - a request occupying the memory module can be considered as stored locally.
- The employed root queue provides Q buffers (size).
- The Bluetree root multiplexer provides 1 pipelined buffer.
- Either the Bluetree multiplexer adjacent to the root stage provides 1 buffer. If buffers from both Bluetree multiplexers are considered, there can be path contention. With the aim to guarantee the *queued service*, only one buffer can be considered as applicable.

With the analysis above, the total size of the queued buffer within the architecture is $Q+3$. By contrast, the total outstanding request number can be assumed as $N_{RQ}(B)$ according to the analysis of memory workload in Section IV-A. Therefore, the minimum size of the root queue Q_S for the *queued service* is $Q_S = N_{RQ}(B) - 3$. The *queued service* requirement can be summarised as follows:

$$Q >= Q_S, where\ Q_S = N_{RQ}(B) - 3 \tag{7}$$

The root queue modification introduces very low overhead. From the perspective of the hardware, to employ the FIFO queue buffers with the appropriate size requires very few extra resources, compared with the entire interconnect. Besides that, this method requires no modification to software operations. When the *queued service* requirement is satisfied, the system stores the outstanding memory requests into the root buffers in sequence. The queue modification effectively smooths the sharing of the critical resource within the multi-core architecture. It reduces memory access latency variation and facilitates timing analysis or verification for real-time applications. This platform supports further software development.

Predictable behaviour: The employment of the root queue introduces additional blocking within the memory architecture. According to the blocking analysis in Section III-B, memory requests stalled in the root queue only leads to *intra-path blocking*. With blocking at the tree root, the entire interconnect will be affected. The request flow in each path stalls. However, this does not complicate the blocking behaviour within the shared memory multi-core interconnect, and the maximum latency due to the queued blocking will increase by an amount proportional to the root memory latency t_D.

The maximum blocking number in the request path N_{RQ}^{WC} can be determined with the similar calculation in Section III-B. The worst-case assumption follows that the system is flooded by memory requests. Memory request ω in *priority path* gives $\omega \in P_j$. Equation (5) and (6) bound the maximum blocking number at each local stage $N_{RQ}^{WC}(\beta_i)$ with local priority $P_j(\beta_i)$. The iterative calculation is then performed from the client to the interconnect root. With the root queue size Q, the maximum blocking number N_{RQ}^{WC} can be determined with the sum calculation that the iterative process result plus Q. With this worst-case assumption, memory requests suffer pessimistic blocking and hence no latency variation.

As for practical applications, the number of memory requests issued to th system will be limited as the path outstanding request number $N_{RQ}(P_j)$. The value of $N_{RQ}(P_j)$ can be determined according to path memory workload pattern (e.g. exact memory access profiles). The alternative method is to determine $N_{RQ}(P_j)$ according to the architectural feature. For example, AXI bus [8] allows only one outstanding request between the master-slave pair. Then the total system outstanding request number $N_{RQ}(B)$ can be determined with the sum calculation. The increasing of $N_{RQ}(B)$ complicates the timing analysis in the original Bluetree architecture, and the detailed analysis requires the accurate knowledge of both the local Bluetree arbiter states and the request flow states. By contrast, with the root queue modification, the value of $N_{RQ}(B)$ can be used to determine the minimum root queue size Q_S with Equation (7) to satisfy the *queued service* requirement.

When the *queued service* requirement is satisfied, the architecture is able to store the outstanding memory requests into the root buffers, waiting for the service of the shared memory module in FIFO sequence. The root queue modification actually smooths the resource sharing, and hence reduces the latency variation. Besides that, the memory requests suffer the same maximum queued delay, and root memory latency can mask the data path latency across the pipelined buffers. Therefore, the latency of memory request ω across the architecture can be bounded as follows:

$$t(\omega) < N_{RQ}(B) \times t_D \tag{8}$$

According to the analysis in Section IV-A, the interval between two successive memory requests $T_{RQ}(P_j)$ also affects the path workload pattern. With a very small interval value such as 1, memory requests will be issued arriving to the interconnect root more intensively. This actually quickly fills the shared root queue. If $T_{RQ}(P_j)$ remains the same value, the memory latency will be identical. In contrast, the varying request interval $T_{RQ}(P_j)$ leads to varying memory latency. Considering the memory workload pattern which is dependent on the response time, the new issued requests arrive at the root queue distributed in time. Such memory requests then suffer various queued delays. Therefore, the memory latency within the *locally arbitrated* architecture only varies with the varying memory workloads, but not due to the resource fairness issue.

VI. Evaluation

This section examines the effectiveness of the root queue modification on the latency variation reduction across the *locally arbitrated* architecture. Our evaluation is based on the 8-client Bluetree memory multi-core architecture, including hardware simulations and FPGA experiments.

A. Hardware Simulation

The initial evaluation is performed by hardware simulations, and the experimental method is similar to Section IV-B. The system is implemented using Bluespec System Verilog (BSV) [24], with simulations on BlueSim simulator. The root memory latency is fixed as a constant $t_D = 20$. A *traffic generator* is employed as a client instead of a processor. It simulates totally 36 memory requests, and memory workload patterns follows the parameters in Section IV-A. In this experiment, the request interval $T_{RQ}(P_j)$ is set as 1, and the path outstanding request number $N_{RQ}(P_j)$ is set as *group c* in Table I.

With the platform setup above, the experimental parameter is the Bluetree root queue size Q. The root queue is implemented using bypass FIFO in Bluespec SpecialFIFOs package [24] [25]. The size of the queue Q is reconfigurable. It increases from 0, 5, to 10. $Q = 0$ indicates the Bluetree architecture with no additional root queue buffers. It evaluates modified Bluetree behaviour with unbalanced path workload.

Figure 8 (a) shows the latency variation within the original Bluetree system with no root queue as $Q = 0$. The experiment shares the same results as in Figure 4 (c). As shown in the graph, the latency in each path is similar from the start period of the simulation, increasing sharply in a very short period of time. The system becomes congested relatively quickly with contention at the shared root memory, with a result that the latency of requests increases to a constant (for a period of time). Essentially, as the workload pattern is dependent on the response time, the rate of the request release drops. Then the latency increase stops in turn. In this way, the Bluetree latency in each path tends to reach the corresponding maximum limit. Obviously, the fixed memory interval leads to the regular latency values. The distribution of the scatters shows the variation in the path outstanding request number. Besides that, the unbalanced path workloads also impact the performance of clients nearby. For example, the latency in path P_2 with $N_{RQ}(P_2) = 1$ is affected by path P_3 with $N_{RQ}(P_3) = 3$. The resource fairness issue harms the performance. With the reduction of the system outstanding request number $N_{RQ}(B)$, the contention to the shared resource reduces. Therefore, the latency decreases in the later period of the simulation.

Figure 8 (b) shows the latency variation in the modified architecture with the root queue size $Q = 5$. Compared with Figure 8 (a), some latency lines coincide in Figure 8 (b). This shows the effect of the root queue modification on balancing the average-case latency. With the root queue, some stalling requests can be stored in the shared FIFO buffers instead of blocking in the interconnect path. In this way, the shared memory component response to blocked requests in sequence. This alleviates resource sharing. In this experiment, the root queue modification benefits some memory paths, and latency in path with heavy workload pattern decreases. For example, the latency in path P_4 with $N_{RQ}(P_4) = 3$ no longer varies between 280 and 240, and the value remains constant at approximately 240. By contrast, latency in path with relatively lower workload increases due to the smoothing effects. For example, the latency in path P_7 with $N_{RQ}(P_7) = 1$ increases from 200 to 240. However, the queue size $Q = 5$ is not enough to buffer all the outstanding requests in the system. As shown in the graph, the latency in either path P_2 or path P_3 fails to coincide with others. With the increasing of the root queue size Q, the balancing effect will be more noticeable.

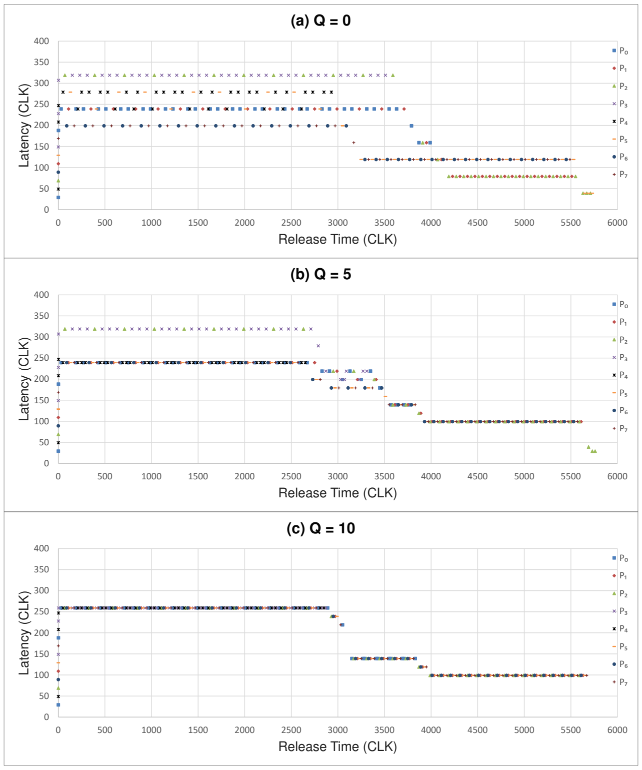

Fig. 8: Latency Variation with Root Queue

When the root queue is reconfigured $Q = 10$ in Figure 8 (c), the latency variation is eliminated. The modified architecture satisfies the *queued service* requirement, $Q >= Q_S$, where $Q_S = N_{RQ}(B) - 3 = 13 - 3 = 10$ in this case. As the request interval is constant, the *traffic generator* releases requests intensively. The root queue is filled in a very short period of time, and then all the latency lines coincide. The latency is actually identical due to the fixed request interval. The worst-case latency can also be bounded as $t_S = N_{RQ}(B) \times t_D = 13 \times 20 = 260$, and the accurate highest observed value is 259 in Figure 8 (c). It drops significantly compared with approximately 325 in Figure (a). In later period of the simulation, latency lines still coincide. The root queue modification effectively reduces the latency variation across the Bluetree memory architecture.

B. FPGA Experiments

This section further evaluates the effectiveness of the architectural modification with synthetic memory workload, and the experimental method is similar to Section IV-B. The evaluation

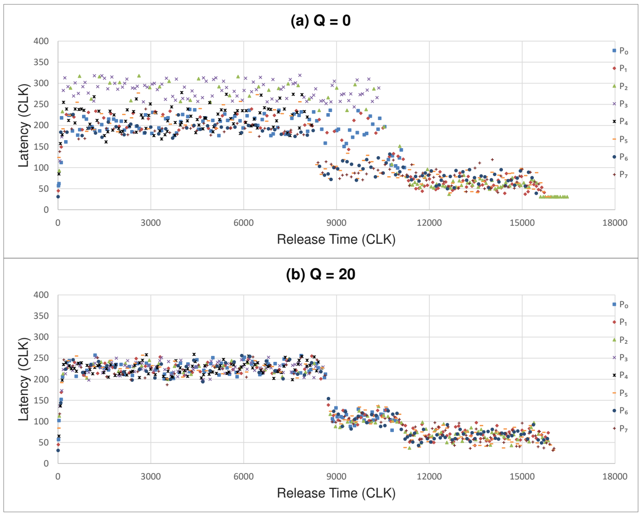

Fig. 9: Latency Variation with Varying Memory Workloads

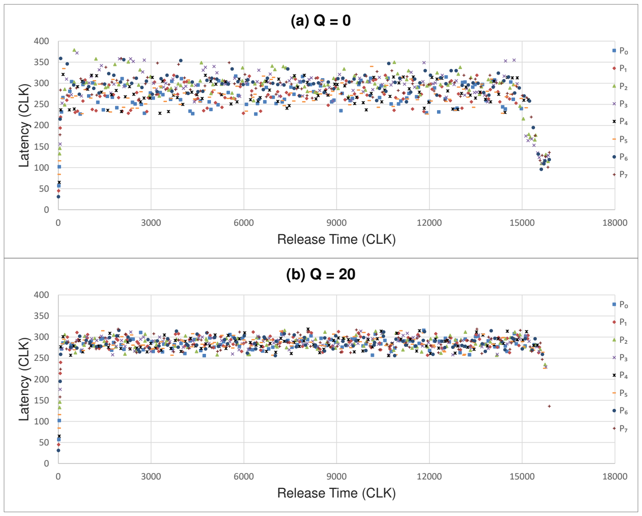

Fig. 10: Latency Variation with Balanced Path Workloads

work is performed by FPGA experiments based on the 8-client system with the root memory latency $t_D = 20$ implemented on Zedboard [26] (using Xilinx Vivado [27] [28]). The *traffic generator* is employed as client. The synthetic memory workload includes the path outstanding request number $N_{RQ}(P_j)$ and the request interval $T_{RQ}(P_j)$. With the platform setup above, the reconfigurable root queue size Q varies as the experimental parameter. The measured metrics are release time and latency. Three groups of experiments are presented.

1) Latency Variation with Varying Memory Workloads: This experiment evaluates latency variation with unbalanced varying memory workloads. The total memory requests number increases to 100. The path outstanding request number $N_{RQ}(P_j)$ remains as *group c* in Table I, and the memory request interval $T_{RQ}(P_j)$ varies as $T_{RQ}(P_j) \in [1, 64]$.

Figure 9 shows the experimental results with the root queue size Q increasing from 0 to 20. As shown in Figure 9 (a), the latency varies with no root queue $Q = 0$. The latency variation follows the similar trend as in Figure 8 (a). The system becomes congested quickly with intensive memory requests. It causes contention to the shared resource, and the latency increases. Essentially, as the workload pattern is dependent on the response time, the rate of the memory request release drops with the congestion. Then the latency increase stops in turn. The distribution of the scatters reflexes the unbalanced path workloads, and the resource fairness leads to varying latencies.

By contrast, with the root queue size increasing to $Q = 20$, the architecture satisfies the *queued service* requirement. Figure 8 (b) shows flat latencies, and highest observed values are reduced. With the introduction of the varying request interval, the accurate latency no longer keeps identical. In this experiment, the latency variation actually reflexes the varying request interval $T_{RQ}(P_j) \in [1, 64]$. The latency variation approximately reduces from 200 to 50, and the highest observed latency approximately reduces from 320 to 250.

2) Latency Variation with Balanced Path Workloads: This experiment evaluates the Bluetree latency variation with balanced path memory workloads. The number of total requests issued in each path is 100, and request interval $T_{RQ}(P_j)$ varies as $T_{RQ}(P_j) \in [1, 64]$. The path outstanding request number $N_{RQ}(P_j)$ is fixed to 2, balanced for each client.

Figure 10 (a) is with no root $Q = 0$, and Figure 10 (b) is for $Q = 20$, which satisfies the *queued service* requirement. With local round-robin arbitration, Bluetree provides relatively fair accesses to the shared memory for all clients. However, the resource sharing issue within the architecture is still noticeable, even with the balanced outstanding requests. As shown in Figure 10 (a), the highest observed latency is approximately 380, and the latency approximately varies between 220 and 380. By contrast, latency lines tend to coincide in Figure 10 (b). With the root queue size $Q = 20$, the highest observed latency reduces to 320, and the latency only varies within 50. The root queue appears the essential architectural complement.

3) Latency Variation with Increasing Request Intervals: Based on the balanced path outstanding requests, this experiment further evaluates the Bluetree behaviour with increasing memory request intervals. The variation of the interval $T_{RQ}(P_j)$ is increased to $T_{RQ}(P_j) \in [1, 256]$.

Figure 11 shows the experimental results. With the increased memory request intervals, the latency varies widely in Figure 11 (a) with no root queue $Q = 0$. By contrast, the effect of the root queue modification is clearly shown in Figure 11 (b). With the root queue $Q = 20$, latency lines tend to coincide with smaller variation, and the highest observed latency also drops to approximately 230. However, the accurate latency still varies widely. For example, within the global time period between 3000 to 6000, the latency can drop to less than 50, while it can also increase to more than 200. The similar tendency can also be frequently observed in Figure 11 (b). Considering the experimental setup $T_{RQ}(P_j) \in [1, 256]$, the system is not sufficiently loaded. This actually alleviates the resource sharing within the architecture. These memory requests can suffer variable queued delays, and the latency only varies due to the varying memory workload. One potential solution to

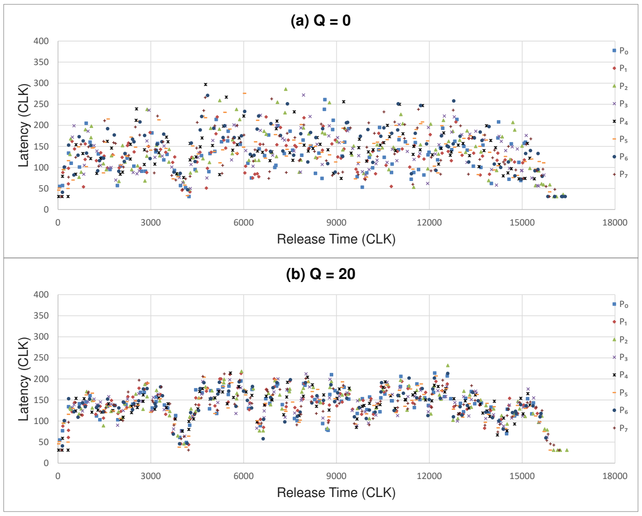

Fig. 11: Latency Variation with Increasing Request Intervals

keep the latency identical can be to utilise dummy memory requests at root of the interconnect.

C. Discussion

Our initial evaluation employs synthetic memory workload. The results show that the root queue modification effectively smooths resource sharing across the *locally arbitrated* architecture. This reduces memory latency variation, and the high latencies are also reduced significantly. The modified hardware platform facilitates timing analysis or verification for real-time applications and better supports software development.

VII. Conclusion

In this paper, we address resource contention for multi-core architectures with distributed memory interconnect. We define the analytical flow for the predictable behaviour of the *locally arbitrated* platform with calculational worst-case bound, and also unveil the latency variation within both *locally arbitrated* and *globally arbitrated* architectures. Then we present the root queue modification to the *locally arbitrated* architecture to smooth resource sharing. Our evaluation shows the effectiveness of this architectural complement that the memory latency variation is effectively reduced across the modified multi-core platform. Further architectural exploration or software co-design development remains future work.

References

[1] L. Benini and G. De Micheli, “Networks on chips: A new soc paradigm,” *Computer -IEEE Computer Society-*, vol. 35, no. 1, pp. 70–78, Jan 2002.

[2] T. Bjerregaard and S. Mahadevan, “A survey of research and practices of network-on-chip,” *ACM Comput. Surv.*, vol. 38, no. 1, Jun. 2006.

[3] M. Caldari, M. Conti, M. Coppola, P. Crippa, S. Orcioni, L. Pieralisi, and C. Turchetti, “System-level power analysis methodology applied to the amba ahb bus,” in *Proceedings of the Conference on Design, Automation and Test in Europe: Designers’ Forum - Volume 2*, ser. DATE ’03. Washington, DC, USA: IEEE Computer Society, 2003, p. 20032.

[4] Xilinx, *AXI Interconnect*.

[5] W. J. Dally and B. Towles, “Route packets, not wires: On-chip inteconnection networks,” in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC ’01. New York, NY, USA: ACM, 2001, pp. 684–689.

[6] H. G. Lee, N. Chang, U. Y. Ogras, and R. Marculescu, “On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 12, no. 3, pp. 23:1–23:20, May 2008.

[7] J. H. Rutgers, M. J. G. Bekooij, and G. J. M. Smit, “Evaluation of a connectionless noc for a real-time distributed shared memory many-core system,” in *2012 15th Euromicro Conference on Digital System Design*, Sep. 2012, pp. 727–730.

[8] ARM, *AMBA AXI and ACE Protocol Specification*.

[9] G. Plumbridge, J. Whitham, and N. Audsley, “Blueshell: A platform for rapid prototyping of multiprocessor nocs and accelerators,” *SIGARCH Comput. Archit. News*, vol. 41, no. 5, pp. 107–117, Jun. 2014.

[10] N. Audsley, “Memory architecturesfor noc-based real-time mixed criticality systems,” *Proc. WMC, RTSS*, pp. 37–42, 2013.

[11] M. Schoeberl, D. V. Chong, W. Puffitsch, and J. Sparsø, “A Time-Predictable Memory Network-on-Chip,” in *14th International Workshop on Worst-Case Execution Time Analysis*, ser. OpenAccess Series in Informatics (OASIcs), vol. 39. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2014, pp. 53–62.

[12] M. Dev Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens, “A generic, scalable and globally arbitrated memory tree for shared dram access in real-time systems,” in *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2015, pp. 193–198.

[13] M. D. Gomony, J. Garside, B. Akesson, N. Audsley, and K. Goossens, “A globally arbitrated memory tree for mixed-time-criticality systems,” *IEEE Transactions on Computers*, vol. 66, no. 2, pp. 212–225, Feb 2017.

[14] A. Sharifi, E. Kultursay, M. Kandemir, and C. R. Das, “Addressing end-to-end memory access latency in noc-based multicores,” in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-45. Washington, DC, USA: IEEE Computer Society, 2012, pp. 294–304.

[15] I. Walter, I. Cidon, R. Ginosar, and A. Kolodny, “Access regulation to hot-modules in wormhole nocs,” in *First International Symposium on Networks-on-Chip (NOCS’07)*, May 2007, pp. 137–148.

[16] Z. Shi and A. Burns, “Real-time communication analysis for on-chip networks with wormhole switching,” in *Second ACM/IEEE International Symposium on Networks-on-Chip (nocs 2008)*, April 2008, pp. 161–170.

[17] A. Hansson, M. Coenen, and K. Goossens, “Channel trees: Reducing latency by sharing time slots in time-multiplexed networks on chip,” in *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, Sep. 2007, pp. 149–154.

[18] N. Kavaldjiev, G. J. M. Smit, and P. G. Jansen, “A virtual channel router for on-chip networks,” in *IEEE International SOC Conference, 2004. Proceedings.*, Sep. 2004, pp. 289–293.

[19] A. Mello, L. Tedesco, N. Calazans, and F. Moraes, “Virtual channels in networks on chip: Implementation and evaluation on hermes noc,” in *2005 18th Symposium on Integrated Circuits and Systems Design*, Sep. 2005, pp. 178–183.

[20] G. F. Pfister and V. A. Norton, “Hot spot contention and combining in multistage interconnection networks,” *IEEE Transactions on Computers*, vol. C-34, no. 10, pp. 943–948, Oct 1985.

[21] M. Schoeberl, S. Abbaspour, B. Akesson, N. Audsley, R. Capasso, J. Garside, K. Goossens, S. Goossens, S. Hansen, R. Heckmann, S. Hepp, B. Huber, A. Jordan, E. Kasapaki, J. Knoop, Y. Li, D. Prokesch, W. Puffitsch, P. Puschner, and A. Tocchi, “T-crest: Time-predictable multi-core architecture for embedded systems,” *Journal of Systems Architecture*, vol. 61, 04 2015.

[22] J. Garside and N. C. Audsley, “Wcet preserving hardware prefetch for many-core real-time systems,” in *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems*, ser. RTNS ’14. New York, NY, USA: ACM, 2014, pp. 193:193–193:202.

[23] D. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[24] Bluespec, https://bluespec.com/.

[25] *Bluespec System Verilog Reference Guide*.

[26] ZedBoard, http://www.zedboard.org/product/zedboard.

[27] Xilinx, https://www.xilinx.com.

[28] Vivado, https://www.xilinx.com/products/design-tools/vivado.html.

[29] Xilinx, *7 Series FPGAs Memory Resources*.