BlueScale: A Scalable Memory Architecture for Predictable Real-Time Computing on Highly Integrated SoCs

Zhe Jiang
ARM Ltd, UK

Kecheng Yang
Texas State University, USA

Neil Audsley
City, University of London, UK

Nathan Fisher
Wayne State University, USA

Weisong Shi
Wayne State University, USA

Zheng Dong[§]
Wayne State University, USA

Abstract

In real-time embedded computing, time-predictability and performance are required simultaneously by memory transactions. However, with increasingly more elements being integrated into hardware, memory interconnects become a critical stumbling block to satisfying timing correctness, due to lack of hardware and scheduling scalability. In this paper, we propose a new hierarchically distributed memory interconnect, BlueScale, managing memory transactions using identical Scale Elements, which ensures hardware scalability. The Scale Element introduces two nested priority queues, achieving iterative compositional scheduling for memory transactions, guaranteeing transaction tasks' scheduling schedulability. Associated with the new architecture, a theoretical model is established to improve BlueScale's real-time performance.

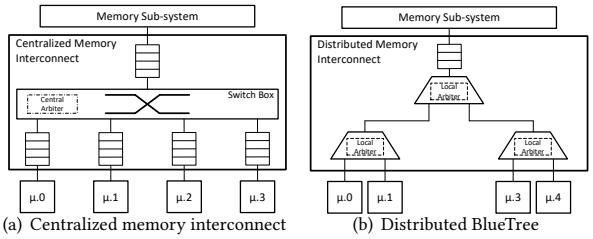
ACM Reference Format:
Zhe Jiang, Kecheng Yang, Neil Audsley, Nathan Fisher, Weisong Shi, and Zheng Dong[§]. 2022. BlueScale: A Scalable Memory Architecture for Predictable Real-Time Computing on Highly Integrated SoCs. In *Proceedings of Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC) (DAC '22).* ACM, San Frncisco, CA, USA, 6 pages. https://doi.org/10.1145/3489517.3530612

1 Introduction

In modern real-time systems, integrating an increasing number of processors and hardware accelerators (HAs) into a System-on-Chip (SoC) is gaining momentum, driven by the diverse functionalities required by modern embedded computing (*e.g.*, image recognition [10]) and the rapid evolution of manufacturing processes in the semiconductor industry (*e.g.*, 5*nm* ASICs production [1]).

The memory sub-system is a vital shared resource in embedded computing architecture, as the execution of clients (*i.e.*, processors and HAs) relies on it heavily [9]. In general, the processing speed of the memory sub-system (a provider) is slower than most clients; hence, the real-time performance of memory processing directly influences the clients' utilization, performance and predictability, *etc.* [9, 20]. As '*bridges*' connecting the memory sub-system and the clients, *memory interconnects* therefore become a dominant factor when determining the entire system's real-time performance [9].

Centralized memory interconnect. In conventional computing architectures, memory interconnects (*e.g.*, AXI-interconnect [2]) often adopt a centralized design (Fig. 1(a)), deploying a monolithic switch box and a central arbiter to buffer and schedule memory transactions. With a global view of memory transactions, centralized interconnects can achieve near optimal real-time performance when a system only has a small number of clients, *e.g.*, a quad-core system [11]. A considerable amount of research, including AXI-hyperconnect [15] and AXI-interconnect$^{\text{RT}}$ (AXI-IC$^{\text{RT}}$) [11], has bounded the time-predictability and performance of the centralized memory interconnect by modifying its micro-architecture and scheduling strategies, *e.g.*, allocating memory bandwidth to a client based on its workload.



(a) Centralized memory interconnect (b) Distributed BlueTree

Figure 1: Top-level architecture of centralized and distributed interconnects with four clients (μ.x: client with ID #x).

However, with an increasing number of clients, the centralized memory interconnect becomes a critical stumbling block, impeding the *scalability* of a real-time system. The reason for this is twofold: from the hardware perspective, with a growing number of clients, the logic size of the switch box and the arbiter increases, limiting the maximum synthesizable clock frequency, leading the interconnect to become the system's critical path [9, 16]; meanwhile, the centralized scheduler has to update the scheduling queue frequently if software tasks on any client are altered. As more clients are involved in the system, the scheduling overhead is not affordable.

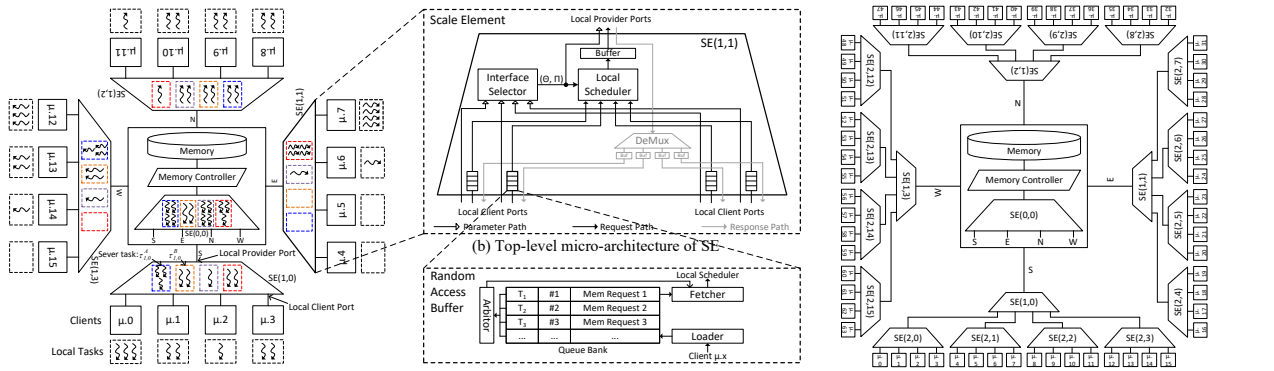
Distributed memory interconnect. To ensure hardware scalability, distributed memory interconnects have recently been investigated, including BlueTree [16], GSMTree [8] and their variants [7, 19, 20]. Unlike the centralized design, a distributed memory interconnect (Fig. 1(b)) restructures the transaction paths based on staged pipelines and employs multiple local arbiters through the pipelines. A local arbiter only manages the memory transactions in a specific section of the pipeline. Since the local arbiters are synthesized separately, the distributed memory interconnects are usually synthesized with a higher clock frequency, achieving better hardware scalability compared to the centralized memory interconnects. However, the distributed design magnifies the issue of scheduling scalability. Specifically, the memory transactions transmitted through a distributed memory interconnect usually involve multiple local arbiters. As more memory requests are issued simultaneously, contentions increase at each shared arbiter, which may cause severe performance degradation with a high local transaction time. On the other hand, the arbitration methods implemented by the local arbiters are naïve heuristics [20], which do not consider the software's real-time requirements. The scheduling decisions made by each arbiter are entirely independent of the demands of the memory transactions issued by the software tasks, and cannot guarantee the transactions' real-time performance. In summary, in modern many-core real-time systems, it is important but challenging to design a memory interconnect to satisfy *scalability* and *time-predictability* simultaneously.

Contributions. In this paper, we propose *BlueScale*, a new memory architecture for multi-/many-core systems, the ensure the SoCs' real-time performance. To this end, we present:

[§]Corresponding author, dong@wayne.edu.

DAC '22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530612



(a) Top-level architecture of *BlueScale* with 16 clients (b) Top-level micro-architecture of SE (c) Micro-architecture of random access buffer (d) BlueScale with 64 clients

Figure 2: *BlueScale* Overview (μ.x: client with ID #x; SE: Scale Element; VE: Virtual Element).

- A hierarchically distributed memory interconnect, managing memory requests using a set of identical Scale Elements (SEs), which ensures hardware scalability (Section 3).
- A new micro-architecture of SEs, realizing two nested priority queues to implement iterative compositional scheduling of memory requests, which achieves simultaneous time-predictability and scheduling scalability (Section 4).
- An interface selection algorithm and the associated analysis framework to determine the best bandwidth for each server task in the compositional scheduling at each SE (Section 5).
- Extensive experiments, including real-world use cases examining overhead, scalability and time-predictability of BlueScale over state-of-the-art memory interconnects (Section 6).

2 BlueTree: A Distributed Memory Interconnect

Before diving into the newly proposed architecture, we first review a SOTA distributed memory interconnect with its performance issues. BlueTree is a distributed real-time memory interconnect presented by Audsley [3], and implemented and upgraded by Garside *et al.* [6], Gomony *et al.* [7, 8] and Wang *et al.* [19, 20], *etc.* BlueTree and its variants have also been integrated into different real-time SoCs, including T-CREST [16] and BlueVisor [12], to ensure hardware scalability and achieve a certain level of predictability.

2.1 Basic Architecture.

Fig. 1(b) illustrates a generalized 4-client BlueTree memory architecture, containing clients, a BlueTree memory interconnect, and a shared memory sub-system. A client can be a processor (either single-core or multi-core) or an HA, marked as μ.x, where x is the client's index. BlueTree employs multiple stages of 2-to-1 multiplexers to create a tree network, connecting multiple clients at the bottom of the network and the shared memory sub-system at the top. This design provides a bi-directional memory access path to each client, *i.e.*, request and response paths. When a client issues a memory request, the request is transferred using the request path, while the memory response returns to the client using the response path. When the number of clients increases, BlueTree scales with more multiplexer stages. The independent synthesis and deployment of the multiplexers ensure the *hardware scalability*.

2.2 Scheduling Strategy and Problems.

In a memory request path, BlueTree introduces a local arbiter in each multiplexer, deciding which memory request to relay to the memory direction (potentially the next multiplexer). The arbitration scheme defines a blocking factor α, determining that every α request from the left-hand side can be blocked by at most one request from the right-hand side; hence the left-hand side can be considered as the local high-priority path, and the right-hand side as the local low-priority path. When α is set to 1, BlueTree becomes a distributed binary tree staging with a local Round-Robin scheme.

Scheduling scalability. As described above, memory transactions transmitted through BlueTree usually involve multiple local arbiters in the request path. As more memory requests are issued simultaneously, the contentions increase at shared local arbiters, which may cause severe performance degradation and high local transaction time [19, 20]. The arbitration scheme depends on the blocking factor, which is defined during the hardware development phase. In other words, it does not consider the real-time requirements of the software at run-time and simply assumes the priority for local transaction paths. This results in the scheduling decisions in BlueTree being completely independent of the memory transactions issued by the software tasks, causing a *severe gap* between the hardware design and the system performance. For these reasons, it is difficult to guarantee the real-time performance of BlueTree's memory accesses. As reported by Garside *et al.* [8] and Wang *et al.* [20], in an 8-client BlueTree, the worst-case response time of a memory transaction is up to 6 times higher than the average case, leading to significant timing variances of memory accesses. Such uncertainty is further increased when integrating more clients into the system, which magnifies the problem of *scheduling scalability*. Although research, such as Gomony *et al.* [7], alleviates the timing variance during memory accesses by allocating time budgets to specific transaction paths (*i.e.*, Time Division Multiplexing (TDM)), they still fail to build links between scheduling and task demands. Additionally, similar to the centralized real-time memory interconnects (*e.g.*, [11, 15]), this work requires recalculation of the memory bandwidth of all clients if the software tasks on any one client are altered, which further degrades the utilization of BlueTree and the memory sub-system [19].

3 *BlueScale*: Overview

In coping with the scheduling scalability issue, we present a new *hierarchically* distributed real-time memory interconnect named *BlueScale*, which employs a set of Scale Elements (SEs) to manage memory requests. SEs are organized as a Quadtree (see Fig. 2(a) and Fig. 2(d)), and each SE has an index of SE(x, y), where x indicates the SE's depth in the Quadtree and y represents its order at this depth. Generally, each SE only needs local information from neighbouring SEs on the request paths (i.e., SE($x+1, 4y$), SE($x+1, 4y+1$), SE($x+1, 4y+2$), SE($x+1, 4y+3$)) to determine which memory request should be processed at each time, while being able to guarantee the real-time performance of the whole system.

3.1 Top-level Architecture

Fig. 2(a) illustrates the top-level architecture of *BlueScale*, which is established using *distributed* 4-to-1 *SEs*, forming a tree connecting the clients and the memory sub-system. The clients are the "leaves of the tree" and the memory sub-system is the "tree root". An SE contains four local client ports and one local provider port, which are connected to its local clients and local provider respectively.

Algorithm 1: BlueScale scheduling under GEDF

input : Ready(t), which is the ready server task set at time t
output: Sched(t), which is the scheduled job at time t

```
Sched(t) = ∅
while (Sched(t) = ∅ & Ready(t) ≠ ∅) do
    Loop through Ready(t) to find the server task τ^X with the
    earliest deadline, where X ∈ {A, B, C, D}.
    if τ^X has local tasks then
        Loop through all local tasks in τ^X to find the local task τ_i
        with the earliest deadline
        if τ_i has a pending job τ_{i,j} then
            Sched(t) = τ_{i,j}
        end
        else
            Remove τ_i from τ^X
        end
    end
    else
        Remove τ^X from Ready(t)
    end
end
```

Based on the locations of an SE, the local client can be a system-wide client (processors and HAs) or another SE. Similarly, the local provider can be the memory sub-system as well as a SE. An SE manages its memory transactions using only local information between its local clients and the local provider, without requiring a global view; hence, it can be synthesized independently.

Scale element (SE). Fig. 2(b) describes the high-level micro architecture of an SE, containing three paths for memory transactions, the request, parameter, and response paths. In the request path, four priority queues are deployed to buffer and prioritize the memory requests sent from each local client. A local scheduler is used to decide from which priority queue the memory requests are transferred to the local provider, assisted by an interface selector in the parameter path. In the response path, a demultiplexer is adopted to route the memory response back to the local clients. Next, we explain how to achieve scheduling scalability by developing a real-time scheduler at each SE, based on the hardware design.

3.2 Scheduling Strategy

In each SE, a compositional scheduler is implemented to schedule memory requests in a hierarchical manner [17]. Based on the proposed architecture, each SE has four local clients. The clients are viewed as isolated components, with each component having the illusion of executing on a dedicated Virtual Element (VE). At runtime, an upper-level scheduler distributes the transaction capacity of the SE to each component and determines the characteristics of the VE in each component. Each component then has a lower-level scheduler to schedule the tasks in that component on that VE. In order to analyze and certify each component independently, interfaces are needed to characterize the supply provided by a VE, *i.e.*, the available transaction time units from the physical SE to support task execution. The periodic resource model [17] is an example of such a fundamental interface. This characterizes a VE using a pair of parameters (Π, Θ), with the interpretation that at least Θ time units of processor time are guaranteed to the supported task set every Π time units. The quotient Θ/Π is called the bandwidth of the VE. Practically, server tasks are introduced to realize the compositional scheduling at each SE, where Π indicates the period of the server task and Θ represents the server task's execution time. The periods and execution times of the server tasks will be determined by an interface selection algorithm (Sec. 5) based on the parameters of the local tasks. Fig. 2(a) shows an example system. For SE(1,0), on the upper-level, four components $\mu.0$, $\mu.1$, $\mu.2$, and $\mu.3$ are supported by four server tasks $\tau^A_{1,0}$, $\tau^B_{1,0}$, $\tau^C_{1,0}$, and $\tau^D_{1,0}$, which are scheduled as four conventional tasks on the SE. Similarly, SE(0,0) also has four components, SE(1,0), SE(1,1), SE(1,2), and SE(1,3), represented by four server tasks $\tau^A_{0,0}$, $\tau^B_{0,0}$, $\tau^C_{0,0}$, and $\tau^D_{0,0}$, respectively. Note that the server tasks executed on SE(1,0), SE(1,1), SE(1,2) and SE(1,3) are considered to be local tasks from the point of view of SE(0,0). Fig. 2(d) shows an example system with 64 clients. Based on the proposed design, *BlueScale* has some useful properties which ensure the scheduling scalability and predictability of the new architecture:

- All the SEs are isomorphic, including the hardware architecture and the real-time scheduler. Thus, the overhead for system implementation and integration is light.
- Scheduling decisions depend entirely on the timing requirements of local tasks, which are obtained from the task parameters.
- When a task joins or leaves a client, the system will only update the parameters of the server tasks on the corresponding memory request path, and all the other server tasks will remain the same; if multiple tasks join or leave the system simultaneously, the SEs involved in the memory request paths can refurbish the server tasks' parameters at the same time, in a distributed manner.

4 Scale Elements: Design

To realize the compositional scheduling (which is described by Algorithm 1), we implement *two nested priority queues* in each SE. In the rest of this section, the hardware architecture of the SE will be introduced, which supports the proposed scheduling method.

4.1 Random Access Buffers

We implemented the low-level priority queue using random access buffers. Unlike conventional FIFO queues, random access buffers have a more complicated micro-architecture (see Fig. 2(c)) to enable *random accesses* of the stored contents. This uses a register chain and register banks to maintain memory requests and their associated parameters. The register chain is physically connected to a loader and a fetcher, and the register banks are connected to an arbiter. The arbiter is implemented using comparators and multiplexers, where the comparators can read the parameters of each request and the multiplexers are connected to the requests' identifiers. At runtime, the comparators continuously check the request parameters in the register banks, searching for the request with the highest priority, and controlling the multiplexers to send the request's identifier to the fetcher. According to the identifier, the fetcher then transfers the corresponding request to the local scheduler.

4.2 Local Scheduler

The local scheduler realizes the upper-level priority queue in an SE. A local scheduler mainly consists of two elements (see Fig. 3(a)):

Server tasks. Server tasks (*i.e.*, τ^A, τ^B, τ^C, and τ^D) are performed using countdown counters, where a Period counter (P-counter) stores the server task's period (Π) and a Budget-counter (B-counter) stores its execution time (Θ). Note that Π and Θ are determined by the interface selection algorithms, which are introduced in Sec. 5. The same micro-architecture is adopted for both counters (see Fig. 3(b)). A counter has two registers which store the counter's reset value and current value. At the counter interfaces, three input ports and one output port are introduced. The input ports are used to program, reset, and enable/trigger the counter, and the output port returns the counter's current value. During run-time, the counter's reset value can be updated by an interface selector using its program port, and the counter's current value is reset when its reset port equals 0. The counter decreases the current value by one when its enable port meets a clock *rising edge*. To refresh a server task (*e.g.*, τ^X) every Π_X with Θ_X time capacity, the reset values in P-counter and B-counter are configured as Π_X and Θ_X, respectively. The P-counter output is connected to its own reset port and that of its associated B-counter.

Scheduling circuits. The scheduling circuits prioritize server tasks. To achieve this, we connected the B-counter output with a constant

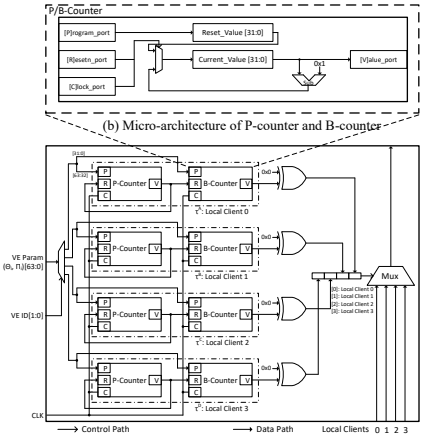


Figure 3: Micro-architecture of the local scheduler.

0x0 using an XOR gate, which checks whether the server task (τ^X) is supplying enough time capacity (Θ_X) for its local client. If the server task has enough time capacity (*i.e.*, $\Theta_X > 0$), a single-bit "1" is returned, otherwise "0" is returned. The scheduling circuits store the results from inspecting the server tasks in a 4-bit register and connect the register to a multiplexer, deciding the memory request from which the local client can be transferred. If more than one server task provides enough time budget at the same time, the memory request with the highest priority is transferred. Note that, as we designed the scheduling circuits using pure combinational logic, scheduling decisions are always made in a single clock cycle.

4.3 Interface Selector

The interface selector calculates the characteristics (Π and Θ) of the server tasks and delivers the calculated results to the interface selector in the next SE, which consists of two elements (see Fig. 4):

Task parameter table. A task parameter table is implemented using a register chain, storing parameters of the tasks executed on local clients. The width of the parameter table is 74 bits, storing client ID (2 bits), task ID (8 bits), period (32 bits) and execution time (32 bits). The depth of the parameter table is customizable. We configured the table depth at 16 for SEs whose local clients are other SEs, since each local client has up to four server tasks.

Computation circuits. The computation circuits contain both data and control paths. The data path has an ALU, a fetcher, and a scratchpad (2 KB). The control path is implemented using a Finite State Machine (FSM) to manage the data flow in the data path to perform the interface selection algorithm.

5 Interface Selection Algorithm and Analysis

Since *BlueScale* is organized as a Quadtree, we index the level of SE depth from 0 to L and there are up to 4^ℓ SEs at level $\ell (0 \le \ell \le L)$. For the sake of a unified analysis framework, we further treat the *Clients* as level $(L+1)$ and treat the *Local Tasks* as level $(L+2)$.

We define the following **interface selection problem at level-ℓ**: *The interface selectors in the level-ℓ SEs select the interfaces for the VEs for level-$(\ell+1)$ elements, given that the interfaces for level-$(\ell+2)$ elements, which are treated as* tasks*, have been known at this point. The problem is to select the* minimum-bandwidth *interfaces for the VEs while ensuring the real-time schedulability of the tasks.*

By resolving the interface selection problems in the reverse order of levels (i.e., from level-L to level-0), all interfaces can be determined. To see this series of problems are well-defined: initially, the interface selection problem at level-L is well defined, because level-$(L+2)$ elements are the *Local Tasks*, of which the parameters are fixed by the application designer; then, for ℓ from L down to 1, upon the completion of the interface selection problem at level-ℓ that determines the interfaces of level-$(\ell+1)$ elements, the interface

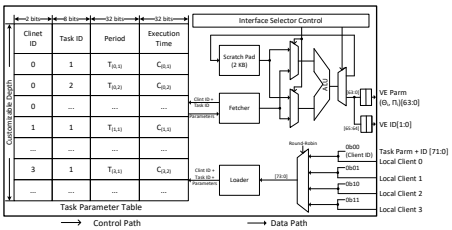


Figure 4: Micro-architecture of the interface selector.

selection problem at level-$(\ell-1)$ is well-defined. In the rest of this section, we only need to focus on a general algorithm that resolves the **interface selection problem at level-ℓ** as defined above.

In particular, we consider each VE X (at level-$\ell+1$) respectively and determine its interfaces (Π_X, Θ_X) such that the bandwidth Θ_X/Π_X is minimized while the real-time schedulability of all tasks (at level-$\ell+2$) that belong to VE X is guaranteed. We denote the set of tasks that belong to VE X by $\mathcal{T}_X$ and denote the set of all tasks at level-$(\ell+2)$ by $\mathcal{T}_{\ell+2}$. It is evident that $\mathcal{T}_X \subseteq \mathcal{T}_{\ell+2}$. Each task τ_i is specified by a pair (T_i, C_i), where T_i is the period as well as the relative deadline and C_i is the worst-case execution time.[1] Note that, we assume discrete time, i.e., T_i, C_i, Π_X, Θ_X are integers. The utilization of τ_i is $u_i = C_i/T_i$. We also denote $U_X = \sum_{\tau_i \in \mathcal{T}_X} u_i$ and $U_{\ell+2} = \sum_{\tau_i \in \mathcal{T}_{\ell+2}} u_i$. The *supply bound function (SBF)* of the VE X, denoted $\mathrm{sbf}(t, X)$, indicates the *minimum* processor time dedicated to VE X during any time interval of length t. Shin and Lee [17] have shown that $\mathrm{sbf}(t, X)$ can be calculated by

$$\mathrm{sbf}(t, X) = \begin{cases} 0 & \text{if } t' < 0 \\ \lfloor t'/\Pi_X \rfloor \cdot \Theta_X + \epsilon & \text{if } t' \ge 0 \end{cases}$$

where

$$t' = t - (\Pi_X - \Theta_X),$$
$$\epsilon = \max\left(t' - \Pi_X \cdot \lfloor t'/\Pi_X \rfloor - (\Pi_X - \Theta_X), 0\right).$$

Meanwhile, all tasks are scheduled under Earliest-Deadline-First policy, thus the *demand bound function* of a task τ_i is denoted by

$$\mathrm{dbf}(t, \tau_i) = \lfloor t/T_i \rfloor \cdot C_i,$$

and the demand bound function for a task set $\mathcal{T}$ is

$$\mathrm{dbf}(t, \mathcal{T}) = \sum_{\tau_i \in \mathcal{T}} \mathrm{dbf}(t, \tau_i).$$

According to [17], $\mathcal{T}_X$ is schedulable (i.e., all deadlines of tasks that belong to VE X must be met), if $\mathrm{dbf}(t, \mathcal{T}_X) \le \mathrm{sbf}(t, X)$ for all t. Furthermore, the following theorem provides finite bound of t to test, in addition to $\Theta_X/\Pi_X > U_X$ which is necessarily required.

Theorem 1. *If* $\mathrm{dbf}(t, \mathcal{T}_X) \le \mathrm{sbf}(t, X)$ *for all* $t < \beta$ *where*

$$\beta = \frac{2\Theta_X}{\Pi_X}(\Pi_X - \Theta_X)/(\frac{\Theta_X}{\Pi_X} - U_X),$$

then $\mathrm{dbf}(t, \mathcal{T}_X) \le \mathrm{sbf}(t, X)$ *for all* t.

Proof. It is equivalent to show the contrapositive that if $\mathrm{dbf}(t, \mathcal{T}_X) > \mathrm{sbf}(t, X)$ for some t, then $\mathrm{dbf}(t, \mathcal{T}_X) > \mathrm{sbf}(t, X)$ for some $t < \beta$. Observing that $\mathrm{dbf}(t, \mathcal{T}_X) \le (\sum_{\tau_i \in \mathcal{T}_X} u_i) \cdot t = U_X \cdot t$ and $\mathrm{sbf}(t, X) \ge \frac{\Theta_X}{\Pi_X}(t - 2(\Pi_X - \Theta_X))$, $\mathrm{dbf}(t^*, \mathcal{T}_X) > \mathrm{sbf}(t^*, X)$ for some t^* implies that $U_X \cdot t^* > \frac{\Theta_X}{\Pi_X}(t^* - 2(\Pi_X - \Theta_X))$, from which we can conclude that $t^* < \frac{2\Theta_X}{\Pi_X}(\Pi_X - \Theta_X)/(\frac{\Theta_X}{\Pi_X} - U_X) = \beta$. □

By Theorem 1, we have a schedulability test for given task set $\mathcal{T}_X$, and given fixed values of parameters Π_X and Θ_X. It is evident that for given task set $\mathcal{T}_X$ and a fixed value of Π_X, the schedulability is monotonically non-decreasing as the budget Θ_X increases. We can use binary search to find the minimum schedulable Θ_X for given task set $\mathcal{T}_X$ and a fixed value of Π_X. The remaining problem is to

[1] Note that, for level-$(L+2)$, i.e., *Local Tasks* requesting memory access, (T_i, C_i) are given parameters from the real-time system; for level-$(\ell+2)$ where $\ell < L$, τ_i is a server task and therefore $T_i = \Pi_i$ and $C_i = \Theta_i$ where (Π_i, Θ_i) are determined via resolving the interface selection problem at level-$(\ell+1)$ with the interpretation that at least Θ_i time units of transaction time is guaranteed to the supported task set every Π_i time units.

Table 1: Hardware overhead (RAM unit: KB; power unit: mW).

	LUTs	Registers	DSPs	RAMs	Power
AXI-IC^{RT}	3,744	3,451	0	0	46
BlueTree	1,683	2,901	0	0	27
BlueTree-Smooth	2,349	3,455	0	0	41
GSMTree	2,443	3,115	0	8	59
MicroBlaze	4,993	4,295	6	256	369
RISC-V	7,433	16,544	21	512	583
Proposed	2,959	3,312	0	10	67

select Π_X. We give a finite range of feasible choices of Π_X by the following theorem so that all feasible Π_X can be enumerated.

Theorem 2. *To ensure VE X's schedulability, it is necessary that*

$$\Pi_X \leq \frac{\min_{\tau_i \in \mathcal{T}_X}\{T_i\}}{2(U_{\ell+2} - U_X)}.$$

Proof. In order to provide real-time schedulability, the bandwidth for each VE clearly must be at least the total utilization of all tasks that belong to that VE. As a result, the total bandwidth for all VEs other than VE X must be *at least* $\sum_{\tau_i \in \mathcal{T}_{\ell+2} \setminus \mathcal{T}_X} u_i = U_{\ell+2} - U_X$. Given that the total bandwidth (at each level) cannot exceed 1, the bandwidth of VE X is *upper bounded* by $\frac{\Theta_X}{\Pi_X} \leq 1 - U_{\ell+2} + U_X$. Observing that $\text{sbf}(t, X) = 0$ for $t \leq 2(\Pi_X - \Theta_X) = 2(1 - \frac{\Theta_X}{\Pi_X})\Pi_X$, we see that $\min_{\tau_i \in \mathcal{T}_X}\{T_i\} \geq 2(1 - \frac{\Theta_X}{\Pi_X})\Pi_X$ is necessary for the schedulability; otherwise, at least the task with the smallest period must miss its deadlines in the worst case. That is, $\min_{\tau_i \in \mathcal{T}_X}\{T_i\} \geq 2(1 - \frac{\Theta_X}{\Pi_X})\Pi_X \geq 2(1 - (1 - U_{\ell+2} + U_X))\Pi_X = 2(U_{\ell+2} - U_X)\Pi_X$ is necessary, which implies that $\Pi_X \leq \frac{\min_{\tau_i \in \mathcal{T}_X}\{T_i\}}{2(U_{\ell+2} - U_X)}$ is necessary. □

Combining all above steps, the schedulable (if any) pair (Π_X, Θ_X) is found with minimum bandwidth Θ_X/Π_X. To confirm the schedulability, we need to verify that after all interface selection problems (in the order of level-L to level-0) are resolved, the level-0 resource (i.e., the memory controller) is not overutilized (by level-1 server tasks). That is, we check whether $\sum_{\tau^X \in \mathcal{T}_1}(\Theta_X/\Pi_X) \leq 1$ holds.

6 Evaluation

Experimental platform. *BlueScale* was built and implemented on a Xilinx VC707 evaluation board, using BlueSpec System Verilog. Additionally, the experimental platform also had 16/64 MicroBlaze processor, two DNN HAs [21], and a shared memory sub-system. The processor was fully-featured, enabling all performance related functionalities (e.g., pipeline and data cache). We adopted FreeRTOS (v.10.4) as the OS kernel for all processors and compiled the software (OS kernels, drivers and user applications) using a MicroBlaze GNU tool-chain. The DNN HA is instantiated with the default settings to execute light-weight DNN tasks. The shared memory sub-system contains a 4GB DRAM module and a memory controller. The system elements were connected using *BlueScale* and a 9×9 mesh type open-source NoC [14], enabling memory accesses and inter-processor communications, respectively. Moreover, We built baseline systems on similar hardware platforms, replacing *BlueScale* with different memory interconnects: **AXI-IC^{RT}** is a centralized real-time interconnect which contains a monolithic arbiter [11]. **BlueTree** [3] and **BlueTree-Smooth** [19] are distributed memory interconnects, which are introduced in Sec. 2. BlueTree-Smooth deploys additional buffers compared to BlueTree at memory access paths in order to smooth transactions [19]. For BlueTree and BlueTree-Smooth [19, 20], we adopted default settings and configured their blocking factor at 2. **GSMTree** is a variant of BlueTree [7], supporting different strategies for memory bandwidth reservation. Following [7], in **GSMTree-TDM**, we reserved equivalent bandwidths for all clients; in **GSMTree-FBSP**, we reserved memory bandwidths proportional to the maximum workloads on clients.

6.1 Hardware Overhead

Experimental setup. We configured all interconnects to support 16 clients and compared their hardware overhead in terms of Look-Up-Tables (LUTs), registers, DSPs, RAMs, power. To evaluate the overhead of *BlueScale* from a system perspective, we also compared *BlueScale* against two general-purpose processors (MicroBlaze and RISC-V). The MicroBlaze processor was fully-featured as described above. The RISC-V processor was implemented based on [13], supporting all the functionalities of the MicroBlaze, as well as multi-branch, out-of-order processing and related functionalities. All components were synthesized using Vivado (v2021.1) [18].

Obs 1. *BlueScale* required slightly more resources than distributed memory interconnects, similar to a centralized interconnect. As shown in Table 1, *BlueScale* consumed more LUTs and power than the other distributed memory interconnects, but similar registers: BlueTree (175.8% LUTs, 114.2% registers, 248.1% power), BlueTree-Smooth(126.0% LUTs, 95.9% registers, 163.4% power), and GSMTree (121.% LUTs, 106.3% registers, 113.6% power). The additional hardware consumption is introduced by the new micro-architecture. Compared to other system elements, *BlueScale* required significantly less LUTs and registers: AXI-IC^{RT} (79.0% LUTs, 96.0% registers), MicroBlaze (59.3% LUTs, 77.1% registers), and RISC-V (39.8% LUTs, 20.0% registers). In addition, the *BlueScale* implementation only required 8KB RAMs and 0 DSP.

6.2 Hardware Scalability

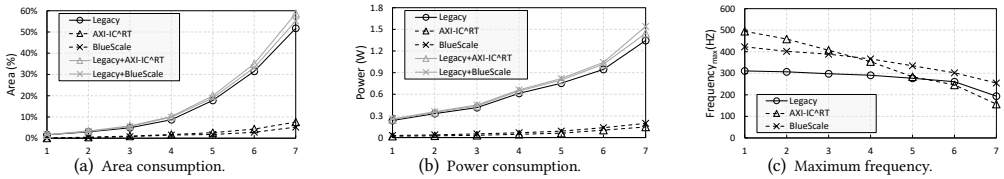
Experimental setup. We adopted the same method described in Sec. 6.1 to implement *BlueScale* and AXI-IC^{RT} with a scaling number of clients (MicroBlaze processors). Additionally, we introduced a scaling factor: η to control the number of clients (2^η). We compared the scalability of area consumption between *BlueScale*, AXI-IC^{RT}, and the corresponding many-core systems with and without them. The area consumption was normalized by the total area of the platform. We then examined the scalability of power consumption, calculated as the sum of static and dynamic power simulated by the tool. Lastly, we evaluated the maximum frequency of *BlueScale* and AXI-IC^{RT} across the legacy systems with different η.

Obs 2. The area and power consumption of *BlueScale* were linearly scaled by η. Compared to a centralized memory interconnect, *BlueScale* consumed less area, but slightly more power. As shown in Fig. 5(a), when the system scaled with η, the area consumption of legacy system, AXI-IC^{RT}, and *BlueScale* consistently increased. In all examined cases, the additionally introduced area consumption was bounded within a small margin – less than 5%. Furthermore, *BlueScale* always required less area than AXI-IC^{RT}, which is aligned with the evaluation of overhead in Obs 1. Power consumption is usually affected by voltage, clock frequency, toggle rate and design area. Since the unified voltage, clock frequency and simulated toggle rate were assigned to all systems, the design area dominated overall power consumption. As shown in Fig. 5(b), power consumption is increased linearly in the systems when η increased.

Obs 3. When the system scaled with η, *BlueScale* did not affect the system's maximum performance. This observation is shown in Fig. 5(c). When the system had more than 32 clients ($\eta > 5$), the maximum frequency of AXI-IC^{RT} became lower than the legacy system, which affected the system's maximum performance. *BlueScale* effectively avoided such issues, as it always achieved a higher maximum frequency than the legacy system.

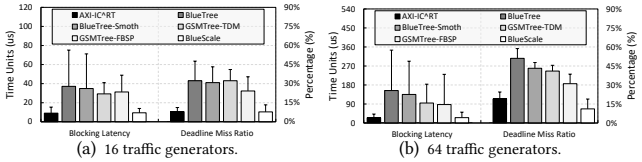
6.3 Interconnect-level Real-time Performance

Experimental setup. We deployed 16/64 traffic generators designed in [20] as clients, simulating memory requests without processing any data. The workloads on the traffic generators were randomly generated offline, with specified periods and implicit deadlines, bounding the interconnect utilization between 70% and 90% in each experimental trial. Each traffic generator had a fixed priority scheduler, with the request priority assigned using GEDF. The synthetic workloads simulated traffic patterns close to practical applications with intensive memory transactions. The experiments are executed 200 times to evaluate two metrics: blocking latency



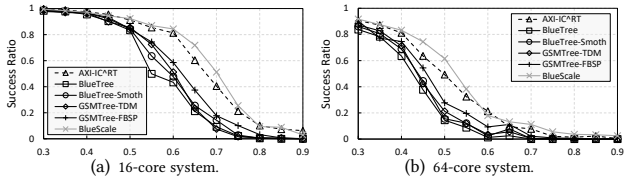
(a) Area consumption. (b) Power consumption. (c) Maximum frequency.

Figure 5: Area, power, and maximum frequency v.s. scaling factor η.



(a) 16 traffic generators. (b) 64 traffic generators.

Figure 6: Synthetic workloads.



(a) 16-core system. (b) 64-core system.

Figure 7: System-level case study (x-axis: target utilization).

and deadline miss ratio. The blocking latency of a request indicates the duration of time it is blocked by requests with a lower priority. The deadline miss ratio records the percentage of memory requests being not completed by the deadlines.

Obs 4. When the system was scaled with various numbers of clients, *BlueScale* achieved the best real-time performance. This observation is summarized in Fig. 6: (i) *BlueScale* always had the shortest blocking time and the highest success ratio; (ii) *BlueScale* always had the least experimental variance. This is because conventional distributed memory interconnects adopt heuristic-based arbitration methods, whereas *BlueScale* and AXI-IC^{RT} support hardware-level prioritization and scheduling of memory requests (see Sec. 3 and 5), ensuring memory requests are transmitted according to their importance, and the system executes predictably.

6.4 Case Study

We configured all examined systems with 16/64 processors and 2 DNN HAs, then executed two sets of real-world tasks: (i) 10 automotive safety tasks, selected from Renesas automotive use case database [5], *e.g.*, CRC, RSA32, core-self test; (ii) 10 automotive function tasks, chosen from EEMBC benchmark [4], *e.g.*, fast Fourier transform, speed calculation, *etc.*. We employed a hybrid-measurement approach to obtain Worst-Case Execution Times for all tasks. Before run-time, the raw data processed by the 20 tasks was randomly generated and stored in the shared memory subsystem. At run-time, the clients used the memory interconnect to fetch the raw data and send the calculated results back to the memory. Each task had a randomly defined period and implicit deadline, with overall processor utilization approximately 30%.

Interference tasks. We used two categories of interference tasks for processors and DNN HAs. The processor interference tasks were selected from the EEMBC benchmark [4], and could be added into the system to control overall processor utilization. Task execution time is affected by diverse factors; hence, adding interference tasks to a processor only gives it a target utilization. The HA interference tasks were built on SqueezeNet architectures, and trained using MNIST, EMNIST and CIFAR-10 training datasets, respectively. The testing datasets used at run-time were pre-loaded into the memory sub-system. Executing interference tasks on HAs intensified memory transactions and made the system heterogeneous. As bandwidth reservation was not supported by all interconnects, we configured the HA to enforce it using only $\frac{1}{\#\_of\_clients}$ of memory bandwidths.

Experimental setup. We introduced two groups of experimental setups, which activated 16/64 processors and a HA to execute the experimental task sets and interference tasks. In each experimental group, we executed each examined system 200 times under varying target utilizations $[10\% - 90\%]$ (at intervals of 5%). Each execution lasted 300 seconds. For fair comparison, we ensured the data input to the examined systems was identical in each execution. We evaluated the examined systems using *success ratio*, recording the percentage of trials that executed successfully (i.e., without deadline misses of any safety or function tasks) under a specified target utilization.

Obs 5. Introducing *BlueScale* was beneficial. This observation is given by Figs. 7(a) and 7(b)). In both 16-core and 64-core systems, *BlueScale* consistently achieved higher success ratios compared to the other distributed memory interconnects. *BlueScale* also outperformed the AXI-IC^{RT} in most experimental trials, which is aligned to the experimental results in Obs.4.

7 Conclusion

This paper proposes a new memory interconnect (*BlueScale*) for real-time SoCs. It employs a distributed hardware architecture to ensure hardware scalability using isomorphic SEs. Each SE adopts two nested priority queues to realize iterative compositional scheduling for memory transactions to achieve guaranteed real-time performance. An interface selection algorithm is derived to determine the best bandwidth for each server task in the compositional scheduling at each SE. Experimental results show that *BlueScale* outperforms state-of-the-art interconnects with varying configurations.

References

[1] SEBASTIAN Anthony. 2017. IBM first 5nm chip. *Ars Technica* (2017).
[2] ARM. 1995. AMBA AXI 5.0. https://developer.arm.com/architectures.
[3] Neil Audsley. 2013. Memory architecture for NoC-based real-time mixed criticality systems. *Proc. WMC, RTSS* (2013), 37–42.
[4] EEMBC. [n.d.]. EEMBC benchmark. https://www.eembc.org/autobench/.
[5] Renesas Electronics. [n.d.]. Renesas: Automotive Use Cases. https://www.renesas.com/solutions/automotive.html.
[6] Garside, Jamie and Audsely, Neil. 2013. Prefetching across a shared memory tree within a network-on-chip architecture. In *IEEE Proc. SoCC*.
[7] Gomony, Garside, Akesson, Audsley, and Goossens. 2016. A globally arbitrated memory tree for mixed-time-criticality systems. *IEEE TC* (2016).
[8] Manil Dev Gomony et al. 2015. A generic, scalable and globally arbitrated memory tree for shared DRAM access. In *IEEE Proc. DATE*.
[9] John L Hennessy. 2011. *Computer architecture: a quantitative approach*.
[10] ISO. 2018. 26262: Road vehicles-Functional safety. *FDIS* (2018).
[11] Zhe Jiang et al. 2021. AXI-InterconnectRT: Towards a Real-Time AXI-Interconnect for System-on-Chips. In *IEEE Proc. RTAS*.
[12] Zhe Jiang and Neil Audsley. 2018. Bluevisor: A scalable real-time hardware hypervisor for many-core embedded systems. In *Proc. RTAS*.
[13] Susumu Mashimo et al. 2019. An Open Source FPGA-Optimized Out-of-Order RISC-V Soft Processor. In *Proc. ICFPT*.
[14] Gary Plumbridge. 2014. Blueshell: a platform for rapid prototyping of multiprocessor NoCs and accelerators. *Computer Architecture News* (2014).
[15] Francesco Restuccia, Alessandro Biondi, Mauro Marinoni, Giorgiomaria Cicero, and Giorgio Buttazzo. 2020. AXI hyperconnect: a predictable, hypervisor-level interconnect for hardware accelerators in FPGA SoC. In *IEEE Proc.DAC*.
[16] Martin Schoeberl et al. 2015. T-CREST: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture* (2015).
[17] I. Shin and I. Lee. 2003. Periodic resource model for compositional real-time guarantees. In *IEEE Proc. RTSS*.
[18] Vivado. [n.d.]. Microblaze. https://www.xilinx.com/download/vivado.html.
[19] Haitong Wang et al. 2020. Addressing resource contention and timing predictability for multi-core architectures with memory interconnects. In *RTAS*.
[20] Haitong Wang et al. 2020. Meshed Bluetree: Time-Predictable Multimemory Interconnect for Multicore Architectures. *IEEE TCAD* (2020).
[21] Dawei Yang. 2021. Nebula: A Scalable and Flexible Accelerator for DNN Multi-Branch Blocks. *Inform. Process. Lett.* (2021).