xBGAS: A Global Address Space Extension on RISC-V for High Performance Computing

Xi Wang
Texas Tech University
xi.wang@ttu.edu

John D. Leidel
Tactical Computing Laboratories
jleidel@tactcomplabs.com

Brody Williams
Texas Tech University
brody.williams@ttu.edu

Alan Ehret
Texas A&M University
ehretaj@tamu.edu

Miguel Mark
Texas A&M University
mmark@tamu.edu

Michel A. Kinsy
Texas A&M University
mkinsy@tamu.edu

Yong Chen
Texas Tech University
yong.chen@ttu.edu

***Abstract*—The tremendous expansion of data volume has driven the transition from monolithic architectures towards systems integrated with discrete and distributed subcomponents in modern scalable high performance computing (HPC) systems. As such, multi-layered software infrastructures have become essential to bridge the gap between heterogeneous commodity devices. However, operations across synthesized components with divergent interfaces inevitably lead to redundant software footprints and undesired latency. Therefore, a scalable and unified computing platform, capable of supporting efficient interactions between individual components, is desirable for large-scale data-intensive applications. In this work, we introduce the Extended Base Global Address Space, or xBGAS, micro-architecture extension to the RISC-V instruction set architecture (ISA) for scalable high performance computing. The xBGAS extension provides native ISA-level support for direct accesses to remote shared memory by mapping remote data objects into a system's extended address space. We perform both software and hardware evaluations of the xBGAS design. The results show that xBGAS reduces instruction count generated by interprocess communication by 69.26% on average. Overall, xBGAS achieves an average performance gain of 21.96% (up to 37.29%) across the tested workloads.**

I. Introduction

Modern high performance computing (HPC) applications such as graph analytics, machine learning, and sparse linear solvers are known to be both memory and data intensive. Large-scale HPC architectures map shared computing and storage resources into discrete nodes to hold enormous data sets that require an increasingly high degree of parallelism. In this scenario, frequent inter-node operations to the shared data in remote nodes induce a significant latency penalty compared to local data operations. Further, additional software infrastructures are required to interface between heterogeneous nodes and devices, resulting in additional overhead.

Existing work performed on leadership class supercomputers such as the Cray T3E, Fujitsu K, IBM Summit, Sunway TaihuLight, and TianHe-series have explored network interconnects and associated enhancements to communication over shared resources [1]–[4]. However, the diverse set of devices that comprise such systems, such as customized processors, domain-specific accelerators, memory devices, interconnects, and network devices are largely architected in a vacuum [5]. The incorporation of these loosely coupled components necessitates multiple convoluted software layers in order to establish connections between disparate devices. This synthesis paradigm not only leads to excessive latency and complexity overheads, but also severely restricts the degree of scalability possible for such HPC systems [6]. As such, a scalable and efficient platform that incorporates distinct components via a high-performance and unified methodology is strongly desired.

Access to remote data in physically distributed memory environments is typically accomplished through the use of the Message Passing Interface (MPI), which distributes and collects data by sending and receiving messages. MPI can optimize workloads that feature readily partitionable datasets and whose memory accesses exhibit a high degree of spatial locality. However, the limited data locality of irregular HPC workloads, such as those utilizing graphs or sparse matrices, are ill-suited to message passing models. As such, the Partitioned Global Address Space (PGAS) paradigm has been introduced to provide efficient one-sided remote data accesses for irregular applications via a memory-semantic programming interface (`put/get`) [7]–[9]. Each PGAS model, including OpenSHMEM, Chapel, Unified Parallel C (UPC), Global Arrays (GA), and CoArray Fortran (CAF), is implemented as either a standalone programming language or a runtime library. However, similar to MPI, the middleware underlying these PGAS implementations implicitly induces superfluous software footprints and associated overheads.

Recent efforts, including those by GenZ, CCIX and OpenCAPI, have applied partitioned address spaces to memory-centric architectures in order to investigate high-performance interconnection models [10]–[12]. These studies found that communication overheads can be effectively reduced by translating miscellaneous device-specific operations and protocols into unified memory operations. However, we have yet to see a generalized architecture with native extended addressing support that is capable of providing efficient communication at the micro-architecture level garner widespread adoption.

To this end, the *Extended Base Global Address Space*, or **xBGAS**, is proposed to provide global, scalable memory addressing support for HPC through the use of a novel extension

1530-2075/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPS49936.2021.00054

to the RISC-V instruction set architecture (ISA). This extension provides up to a 128-bit extended address space to support object-based, flat, or partitioned virtual addressing schemes across multiple distinct nodes. Further, xBGAS leverages the extensible nature of the RISC-V architecture to integrate a set of extended registers and instructions. These resources are then utilized to eliminate software overheads in remote shared data accesses.

In this paper, we introduce our xBGAS research in detail and make four main contributions. First, we present a methodology to create a scalable global address space extension for high-performance communication between distributed shared resources. Second, we detail the xBGAS micro-architecture design, as well as the ISA-level xBGAS instructions and register file, based on the extensible RISC-V ISA for efficient inter-node data operations. Third, we design a new xBGAS programming model and associated runtime library that provides a high-level programming interface as well as abstractions for distributed shared data objects. Finally, we introduce the xBGAS toolchain and implementations using both software simulators and FPGA-based emulation. We validate and provide a performance evaluation of the xBGAS design with benchmarks and applications representing popular data operations in HPC applications. The xBGAS design and toolchains are open-source and available on GitHub to facilitate the research of the community [13]–[18].

The remainder of this paper is organized as follows. Section II provides background and motivation for this research. Section III introduces the xBGAS extended registers, instructions, addressing model, and micro-architecture design. Section IV showcases the xBGAS runtime library as an interface for the proposed programming model. Section V details the xBGAS toolchain design, as well as associated implementations, in both software and hardware. Section VI discusses the xBGAS experimental results. Finally, we summarize our observations and conclusions in Section VII.

II. Motivations and Related Work

According to the Top500 list of the supercomputers, Summit has over 2.6 PB of DRAM and would require over 52 address bits if all the DRAM resided in a single address space [19]. Furthermore, current exascale systems research projects the imminent emergence of 100PB memory systems, which will require 57 bits of address space. The integration of dense non-volatile memories and fast interconnects also drives a demand for even larger memory spaces. Based on historic rates of growth, it may become necessary to expand common address space sizes before 2030 [20]. Therefore, the RV128 project [20] has been proposed as a 128-bit address space extension to the RISC-V ISA. Rather than customizing 64-bit ABIs, operating systems, register widths, and data paths to build a flat 128-bit address space in a manner similar to RV128, xBGAS enhances addressing capabilities by mapping data objects into an extended address space for efficient cross-node data accesses in large-scale HPC systems. This section introduces the motivation of the xBGAS design and compares it with related works from three orthogonal perspectives, including software overhead, scalability, and generalizability. We detail each perspective in the following subsections.

A. Software Overhead

The synthesis of distributed hardware components largely relies on software bridging. Alongside the growth of data volumes, shared resources are expanding to meet the performance requirements of modern HPC applications. As such, it has become necessary to append an ever-growing number of software layers and frameworks to large-scale HPC systems to bridge these shared data objects. However, the software overhead resulting from the interaction of these non-uniform protocols and APIs results in diminishing returns for performance enhancements. Although many endeavors have been devoted to optimizing inter-node shared memory accesses [7], [8], [21]–[23], performance gains have been critically hindered by the limitations of communication libraries and overheads associated with software and layers of protocols. For example, OpenSHMEM implementations typically rely on some combination of complex software infrastructures, such as the Process Management Interface Exascale (PMIx), Unified Communication X (UCX) framework, Message Passing Interface (MPI), Universal Common Communication Substrate (UCCS), Remote-Direct Memory Access (RDMA), and other network frameworks to facilitate low-level communication. These combined software layers induce significant overheads and performance degradations.

In order to demonstrate the impact of the aforementioned software overhead, we profiled OpenSHMEM `get` operations built upon a UCX 1.6.0 implementation and utilized the UCS profiling tool [24] to analyze the proportion of network I/O latency and software overhead, respectively. The test environment is reported in Table III in detail. As shown in Figure 1, the proportion of network I/O latency increases from 54.92% to 99.10% as the payload size grows from 1B to 1MB. This observation implies that, in the case of large requests (e.g. greater than 1MB), the impact of the software overhead is insignificant as the network transfers of large requests dominate the overall latency. These large network transfer requests are often observed in *regular* workloads with good data locality. However, many modern HPC applications exhibit *irregular* workloads that often access pointer-based data structures such as graphs, unstructured grids, sparse matrices, etc. This trend leads to many irregular, fine-grained (e.g. 1B∼8B) remote shared memory accesses [25]–[27]. In these cases, the software overhead can occupy over 44% of the execution time.

As a validation of the irregular request distributions, we captured memory footprints of the scatter operation `a[b[i]] = c[i]` with 8 threads, where the array `a[]` is shared by thread 0 and indices `b[i]` randomly span over the global shared memory space. As shown in Figure 2, the scatter remote accesses within a randomly selected time window (10,000 cycles) are sparsely distributed, which renders the request aggregation techniques used in [7], [8] impractical.

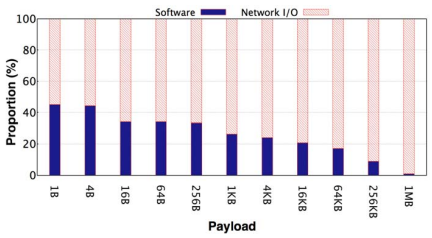

Fig. 1: OpenSHMEM Get Profiling

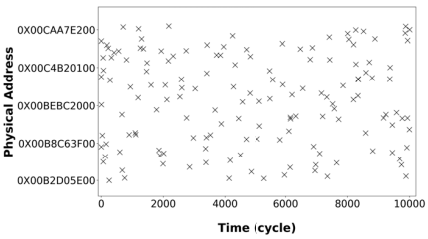

Fig. 2: Random Request Distribution

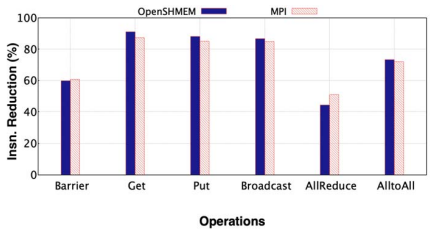

Fig. 3: xBGAS Inst. Reductions

As a result, a tremendous number of small remote accesses expose the overhead of bloated software infrastructures and significantly reduce system performance. However, extended ISA-level instructions possess innate potential to effectively eliminate the overhead of small remote requests.

In order to quantify the software overhead of communication in conventional distributed memory programming paradigms, we measured the instruction counts of 7 widely used operations in HPC applications with OpenSHMEM 3.0.4, OpenMPI 4.0.1, and xBGAS. We then derived the proportion of instructions reduced through the use of the xBGAS, which provides ISA and micro-architecture support for remote data accesses. All these operations were forced to transfer 8B data blocks. As shown in Figure 3, xBGAS dramatically eliminated redundant instruction execution using its native ISA-level support for inter-node communication. On average, 73.89% and 73.46% of the instructions executed by OpenSHMEM and MPI, respectively, can be avoided by utilizing the xBGAS. This series of tests reveal the potential impact of architecture-level optimization on the performance of HPC applications.

B. Scalability

The aforementioned software overhead hinders not only the system performance, but also the integration of more distributed shared resources. As a result, bloated software infrastructures significantly hamper system scalability.

In an effort to enhance scalability and alleviate software overheads, previous works have incorporated architectural features in shared memory systems. For example, the Cray T3D/E architectures introduced extended registers, a DTB Annex, and global segment translations to allow shared memory operations across up to 2048 processors [28], [29]. Scale-Out NUMA (soNUMA) [30] provides a programming model, communication protocol, and architecture for low latency RDMA transfers in a rack-scale system. soNUMA reduces the overall latency of RDMA operations by placing the RDMA controller within the processor's coherent cache hierarchy. Enabling cache coherence between the RDMA interface and CPU allows data received by the RDMA to be placed within the cache hierarchy for fast subsequent accesses. However, buffering remote accesses in the local cache hierarchy can result in cache thrashing and degrade performance of local data operations. Moreover, soNUMA only provides low latency RDMA reads within a single rack as shown in Table I.

In contrast, the xBGAS extension focuses on large-scale systems (e.g. an entire data center) by mapping shared data blocks into the extended scalable address space and utilizing ISA-level remote data operations to directly access the shared data of any node. Additionally, xBGAS and previously proposed architectural techniques that improve shared data access in multi-node systems are not mutually exclusive. For example, the xBGAS ISA extension's bulk transfer interface (discussed in Section IV) can be used to further reduce the latency and related overhead in controlling RDMA engines while making RDMA transfers transparent to the programmer.

C. Generalizability

In some cases, such as with the Cray T3D, T3E, and SGI UV series HPC systems, extending a given micro-architecture to support scalable addressing capabilities sacrifices the generalizability of said architecture. Herein, proprietary instruction sets and customizations inherent in specific processor architectures hinder the adoption of these designs in the modern HPC systems. For example, the SGI UV 300 relied upon a proprietary NUMALINK interconnect and commercially available Intel CPUs to scale up to 64 sockets and 64 TB of memory. In contrast, our proposed xBGAS methodology utilizes an extended microarchitecture that expands the scalability of effective memory spaces over any potential interconnect. The Cray T3D/E, moreover, required specific versions of the UNICOS operating system [31] to support their extended addressing model. It is noteworthy that utilizing an extended addressing model based on a customized OS also necessitates changes to the standard application binary interface (ABI) and existing applications. As a result, such customizations to the OS and ABI circumscribe the generality and portability of these designs.

A flat 128-bit address space, such as the one proposed by RV128, similarly requires OS and ABI modifications in addition to its doubled register width and data path. However, as the RV128 design and specification have been neither finalized nor officially released by the RISC-V community, it is not included for comparisons in Table I.

Driven by the growing need for generalized design, reusable and extensible ISA architecture frameworks have reemerged as a promising solution. As an example, the RISC-V [32] and OpenPiton [33] projects have drawn remarkable interest from both academia and industry by introducing an open-source hardware instruction set architecture and many-core

TABLE I: Comparisons of T3D/E, soNUMA and xBGAS

Project	Scalability	Customized OS	Customized ABI
T3D/E	2k Processors	Required	Required
soNUMA	Single Rack	Not Required	Not Required
xBGAS	Data Center	Not Required	Not Required

research framework, respectively. These generalized instruction set frameworks shed a new light on the modern micro-architecture design, as employed in the GoblinCore-64 (GC64) data-intensive architecture, the PULPino low-power SoC, and the Sanctum software isolation for system security [34]–[36].

The xBGAS extension focuses on the convergence of scalable HPC and extensible architecture techniques, building on knowledge gained from these pioneering efforts. Distinct from RV128 and the Cray T3D/E, the xBGAS provides a generalized extended addressing methodology without the necessity of a customized OS or 64-bit ABI, as presented in Table I.

III. xBGAS Design and Philosophy

Driven by the aforementioned motivations, the xBGAS is designed to enhance the performance of remote data accesses by mapping remote resources into a system's extended address space. The xBGAS extension categorizes memory operations into two basic types: **local** and **global (remote)** operations. *Local* requests refer to data accesses performed using the instructions of the base instruction set. *Global* or *remote* requests denote operations that utilize the xBGAS extended addressing capabilities to access data on remote nodes or storage resources. These remote resources can take the form of a diverse number of storage mediums or partitioning schemes, such as distributed memory systems, block devices, memory mapped file systems, etc.

We use the term **object** to refer to data accessed with global memory operations. Data objects may reside in the physical memory of local or remote nodes. In this section, we describe the xBGAS design, including extended registers, addressing model, micro-architecture, and ISA design.

A. xBGAS Registers

The xBGAS design introduces 32 "extended" registers that are discernible by use of the letter "e" and designated as $e0 \sim e31$. The permissible indices of the extended xBGAS register file are mapped in the same manner as the base general purpose registers (GPRs) of RISC-V: $x0 \sim x31$. The xBGAS registers are also configured using the standard register width (termed **XLEN** in the RISC-V vernacular), which is consistent with the base registers. The XLEN is equivalent to 32 and 64 bits in the RV32I and RV64I instruction sets, respectively. Notably, the extended registers can only be accessed by xBGAS instructions that manage remote data operations. As a result, the xBGAS extension maintains full compatibility with the base ISA and is capable of executing unmodified RISC-V binaries. As such, xBGAS-enabled devices are able to boot and execute RISC-V Linux along with all its ancillary kernel modules without any issues.

B. xBGAS Addressing Model

The xBGAS extension provides scalable addressing capabilities for up to $2 \times XLEN$ bits for any RISC-V system. As such, 64-bit and 128-bit address spaces are available for xBGAS extensions based on RV32I and RV64I, respectively. The xBGAS extension utilizes a compound addressing model to assemble extended global addresses. In the RV64I instruction set, for example, in addition to the standard 64-bit address (*bit[63:0]*) stored in the base GPR, extended registers are configured to hold a unique **namespace** that serves as an object ID, representing a remote resource, as the upper 64-bit address (*bit[127:64]*). Thus, through the utilization of paired base and extended registers, 128-bit addresses are attained to access remote objects. In this way, we can convey requests or responses to the correct destination by specifying the upper 64-bit address (namespace) of a target while utilizing the lower 64-bit address in a traditional manner to locate target data within a remote node.

Notably, the xBGAS extension has no influence on local data accesses. Given that remote objects can be heterogeneous, customizing the local data access policy will inevitably jeopardize the generalizability and compatibility of the xBGAS design. Therefore, we ensure that only xBGAS instructions can access the extended registers and a namespace ID of 0 always denotes a local memory access. As a result, the extended address space is simply ignored by local data operations, regardless of whether or not the xBGAS extension is enabled.

C. xBGAS Architecture Design

Figure 4(a) depicts an overview of the xBGAS architecture design, where each xBGAS processor is extended with two additional hardware components: the **Arbiter** and the **Namespace Lookaside Buffer** (NLB). The arbiter is directly attached to the CPU to route data requests based on the type of operation (local or global). Local accesses are directed to the local memory system to be handled in the conventional manner while the global accesses are forwarded to remote nodes using the extended addresses and the NLB. In order to minimize the impact on the processor's critical path, the arbiter is kept simple. As presented in Figure 4(b), the address and data buses used to issue memory operations are connected to both the L1 cache and memory interface. Separate read and write enable control signals are generated for local and global memory operations. Data signals returned by the L1 cache and network interface are multiplexed based on the type of pending operations. This multiplexer is the only logic added on the data path of the xBGAS core. As instructions are only fetched from local memory, instruction requests bypass the arbiter and proceed directly to the instruction cache. We employ a directory based cache hierarchy in the xBGAS design to maintain data coherency of the local cache hierarchy. Requests from both local and remote sources access a node's local memory through the directory. The directory does not maintain coherency between nodes. Therefore, intra-node data consistency model will not circumscribe the scalability of the entire system. Figure 4(b) shows how local and remote memory

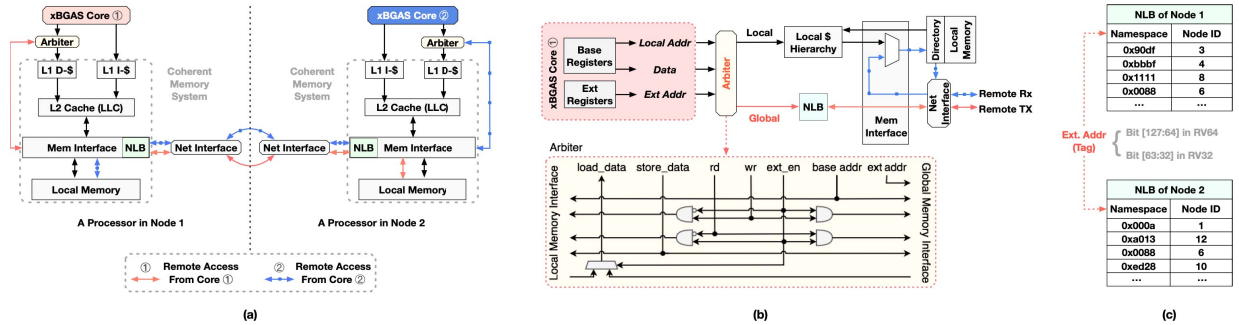

Fig. 4: The xBGAS Architecture Design

I-Type				
Mnemonic	Base	Funct3	Dest	Opcode
eaddie extd, rs1, imm	rs1	111	extd	111˙011
eld rd, imm(rs1)	rs1+ext1	011	rd	111˙011

S-Type				
Mnemonic	Src	Base	Funct3	Opcode
esd rs2, imm(rs1)	rs2	rs1+ext1	011	1111011
esw rs2, imm(rs1)	rs2	rs1+ext1	010	1111011

R-Type						
Mnemonic	Funct7	RS2	RS1	Funct3	RD	Opcode
erld rd, rs1, ext2	1010101	ext2	rs1	011	rd	0110011
ersd rs1, rs2, ext3	0100010	rs2	rs1	011	ext3	0110011

Fig. 5: Encoding Examples of xBGAS Instructions

requests access the directory and local memory. Unlike the soNUMA design [30], which caches remotely accessed data in local caches, xBGAS routes remote requests directly to the memory interface to eliminate the risk of local cache thrashing in the presence of frequent inter-node communication.

The NLB is a fully associative cache with a "least recently used" replacement policy that contains a mapping between recently used namespace IDs and remote node addresses. Each processor's NLB is contained within the memory interface. Figure 4(c) illustrates two NLB examples with namespace mapping. Global operations initiated by the local core are received by the memory interface's NLB. The extended address is utilized as a tag to look up the remote node address associated with the namespace. After the remote node address is known, the global request is sent to the network interface for transmission. The address stored in the base register is then used to access the target data within the remote node.

When a NLB miss occurs, the namespace table in main memory is searched for the correct translation to update the NLB. Systems with large namespace tables may use a hierarchical and distributed scheme so that the whole table does not need to be stored on each node. An initial namespace table is distributed to each node by the xBGAS runtime and updated whenever shared data is allocated or deallocated. An inter-node data flow example is also illustrated in Figures 4(a) and (b). The xBGAS core ① first routes a remote request to the memory interface via the arbiter. The request looks up the remote node address in the NLB and then proceeds to the network interface. This dispatched operation is routed to the memory interface of target node 2, where data is either read or written. Once the operation is complete, the associated data or response is returned via the same path back to xBGAS core ①. Accesses from xBGAS core ② follow an analogous path to access data residing in node 1. Notably, xBGAS employs efficient one-sided communication that involves neither remote CPUs nor system calls.

D. xBGAS ISA Extension

In order to access remote memory, xBGAS includes instructions to support the extended addressing capabilities based on the standard RISC-V Instruction Set Architecture (ISA). Overall, the extended xBGAS instructions can be classified into four categories as described in the following sections.

1) Address Management Instructions: Since existing data movement operations in the base RISC-V instruction sets, such as *addi* and *sd*, cannot operate on the xBGAS extended registers, xBGAS provides three I-type address management instructions: *eaddi*, *eaddie*, and *eaddix* to manage the extended addresses. The address management operations read an extended or general purpose register (GPR), sum the register value with the instruction's immediate value and write back the result to a general purpose or extended register. For example, the instruction *eaddie* stores the sum of a base register (rs1) and a 12-bit immediate value into an extended register (extd). Encoding examples of *eaddie* are illustrated in Figure 5. The detailed encodings of each xBGAS instruction can be found in the xBGAS specification [37].

2) Integer Load/Store Instructions: Similar to the base load and store instructions of the standard RISC-V ISAs, the extended integer load and store instructions are also encoded using the I-type and S-type formats, respectively, to allow the utilization of immediate operands as address offsets. The xBGAS introduces 14 extended load and store instructions such as *eld*, *esd*, *esw*, etc., supporting 1B∼8B operands, representing data sizes: *byte*, *half word*, *word*, and *double word*. We show the mnemonics of *eld*, *esd* and *esw*, as well as the corresponding opcodes and function codes, in Figure 5.

In each instruction, a GPR (rs1) contains the base address, which will be added with a sign-extended 12-bit immediate operand to form the lower 64-bit address (*bit[63:0]*). The upper 64-bit address (*bit[127:64]*) is placed in an extended register (ext1). As there is no encoding space for the extended registers in the I-type or S-type instruction formats, the chosen

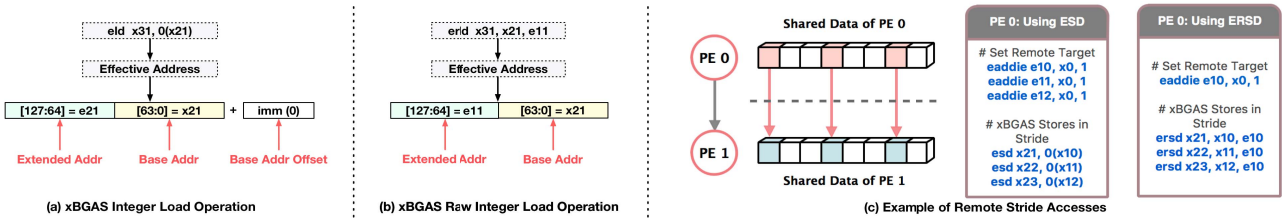

Fig. 6: xBGAS Instruction Examples

extended register corresponds to the index of rs1. Therefore, if the base address of an xBGAS integer load/store instruction is in base register *xN*, extended register *eN* will be utilized for the higher 64-bit address. An example of *eld* is shown in Figure 6(a), wherein the register *x21* holds the base address. Since the index of the utilized base register is 21, the respective extended register *e21* is automatically accessed to obtain the extended address. Given the immediate value is zero, the 128-bit address is synthesized via combining the upper and lower 64-bit addresses stored in registers *e21* and *x21*, respectively.

3) Raw Integer Load/Store Instructions: Besides the load/store instructions detailed above, 12 raw integer load and store instructions are introduced to access remote data, where the extended register containing the upper 64-bit address is explicitly specified. The encoding examples of the double-word raw load and store instructions, *erld* and *ersd* are illustrated in Figure 5, respectively. Figure 6(b) shows an example of *erld* that explicitly specifies the extended register *e11* to assemble the 128-bit address with base register *x21*.

This xBGAS instruction type can eliminate redundant register operations and improve the performance of advanced operations. For instance, Figure 6(c) shows an example of strided operations between PEs 0 and 1. Suppose that the namespace of each PE's shared memory is identical to its PE ID and that there exists a pair of shared arrays symmetrically allocated in PEs 0 and 1, respectively. As PE 0 initiates three remote store operations to the shared data of PE 1 with stride size 3, the requested data of PE 0 is mapped into its registers $x21 \sim x23$ and the corresponding lower 64-bit addresses are stored into registers $x10 \sim x12$. By executing the *ersd* instruction to perform the remote store operations, we only need to set the extended address to 1 (the namespace of remote shared memory) in a single extended register which can then be repeatedly reused. However, if forced to use the *esd* instruction, we need two more *eaddie* instructions to store the same namespace into additional extended registers ($e11$ and $e12$), which results in 50% more redundant instructions for the strided operations as compared to using *ersd*.

4) Remote Atomic Instructions: In addition, xBGAS also introduces extended atomic support to perform remote read-modify-write operations, such as *fetch-and-add*, *compare-and-swap*, etc., wherein each corresponds to a standard RISC-V local atomic operation. In pursuit of efficient one-sided communication, xBGAS atomic requests are offloaded to the network interface controller (NIC) cores and executed without involving the host processors. The associated global atomicity and micro-architecture designs are discussed in [38].

IV. xBGAS Runtime Library

Based on the SPMD (single program, multiple data) programming model, we design the xBGAS runtime library with a simple, yet effective, programming interface that manages remote data accesses and symmetric shared memory.

The xBGAS runtime library provides developers with APIs for one-sided *put* and *get* operations as well as remote atomic and collective operations (*barrier*, *broadcast*, *reduction*, etc.). When accessing a single (register-width) remote data element, the *single-element transfer interface* is invoked, which in turn triggers an extended xBGAS load or store instruction to complete the remote access. Correspondingly, we provide a *bulk transfer interface* to optimize throughput by aggregating remote accesses rather than repetitively issuing xBGAS load/store instructions in a loop to handle large requests. We analyze the performance of this bulk transfer interface in Section VI-B. In addition, the xBGAS runtime also manages the metadata of each PE and shared data objects for the inter-node communications.

V. Toolchain and Implementation

A. xBGAS Toolchain

We have implemented the extended xBGAS registers and instructions in the xBGAS compilers based on the GNU 8.3.0 and LLVM 8.0.0 toolchains [13], [14]. In order to ensure correct binary generation, we also designed an xBGAS assembly test suite [15] that ensures cross-toolchain stability. In addition, we implemented a light-weight runtime library [17] using ANSI C that provides the API used to realize the xBGAS programming model.

B. Software Implementation

We have extended *Spike* [18], a RISC-V simulator, to provide the capabilities necessary for executing the xBGAS instructions. In order to simulate the partitioned global address space and inter-node communication, we integrated MPICH 3.2 into Spike to handle remote memory traffic. Furthermore, we also extended the cycle-accurate Structural Simulation Toolkit (*SST*) 8.0.0 [39] to gather precise runtime statistics related to the xBGAS program executions. The xBGAS implementation and toolchain are open-source and available on GitHub [13]–[18].

TABLE II: xBGAS HW Overhead Against the BRISC-V

Resource	Baseline	xBGAS	Overhead
Logic utilization (ALMs)	1,088	1,783	63.87 %
Total registers	586	1798	206.82 %
Non-regFile registers	586	774	32 %
Total block memory bits	2,048	3,072	50 %
Fmax	78.7MHz	77.7MHz	1.2 %
Average IPC	0.63	0.63	0 %

TABLE III: Simulation Environment Configurations

Parameters	Configurations
Base ISA	RV64I
Node & Core	64 Nodes, 1 Core/Node, 2 GHz
CPU $	8-Way, 16-KB L1, 8-MB L2
NLB	Fully associative, 16 KB, 512 Entries
Memory	DDR4, 2 GB per Node
Network	2D-meshed NoC, 32-bit FLIT

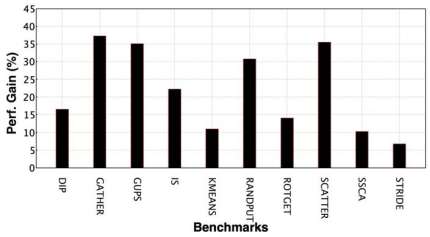

Fig. 7: Performance Gain

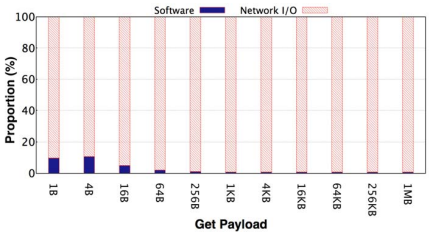

Fig. 8: xBGAS Get Profiling

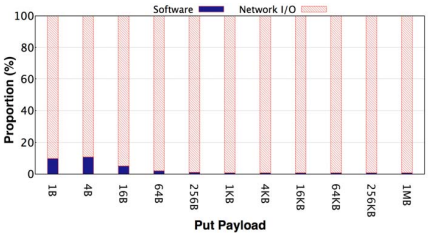

Fig. 9: xBGAS Put Profiling

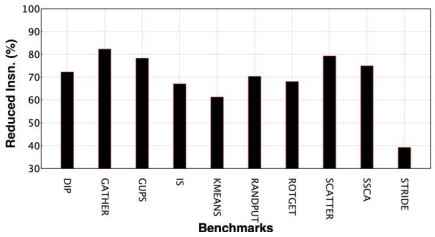

Fig. 10: Instruction Reductions

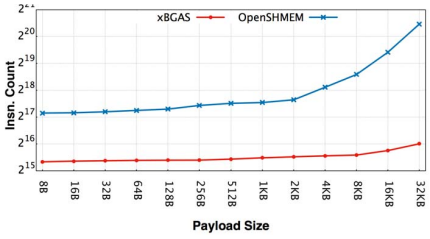

Fig. 11: Broadcast Instruction Count

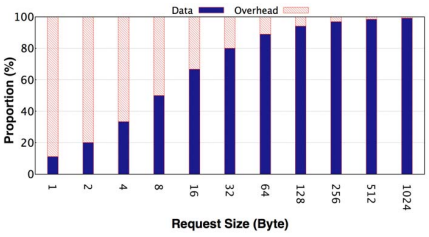

Fig. 12: Bandwidth Utilization

C. Hardware Implementation

We used the open-source BRISC-V Design Space Exploration Platform [40] as a baseline RISC-V hardware implementation. The xBGAS ISA extension is added to the platform's seven stage pipelined in-order core. The xBGAS core includes the extended register file and additional control logic to initiate extended load and store operations. The Node IDs are mapped 1:1 with network addresses. The xBGAS network interfaces connect to NoC routers. The NoCs of individual chips are connected with direct links between remote nodes. NoC routers treat local and remote operations identically, with remote operations routed to chip-to-chip connections instead of local memory controllers. Directly connecting chips together allows load/store messages to be constructed entirely in hardware. We have excluded the cache hierarchy and network on chip (NoC) resource utilization from comparisons presented here because significant modifications to the baseline design are contained within the processor core. All synthesis results are based on a Cyclone V (5CSEMA5F31C6) FPGA. Synthesis is performed using Quartus Prime version 18.0.0. The suite of benchmarks and demonstration programs provided with the BRISC-V Platform are used to calculate an average IPC (Instruction Per Clock) for the baseline and xBGAS cores. Table II presents a comparison of hardware resource usage for the xBGAS core and its baseline BRISC-V core. The HW overhead statistics are post place-and-route. Such overhead is mainly determined by the bit width of the extended registers and the extended physical addresses, regardless of the system size or number of endpoints.

VI. Evaluation and Analyses

A. Software Simulation

1) Benchmarks and Environment: In order to evaluate the efficacy of xBGAS, we select 10 benchmarks, including the OpenSHMEM micro-benchmarks, Oak Ridge OpenSHMEM Benchmarks, Scatter and Gather benchmarks, Scalable Synthetic Compact Applications (SSCAv1), and NAS Parallel Benchmarks (NAS-PB) [41]–[44]. The detailed configuration of the testing environment is listed in Table III. We compiled the aforementioned test suites using the RISC-V GCC 8.3.0 compiler and simulate them on the RISC-V Spike and SST simulators to compare the xBGAS performance against OpenSHMEM. For OpenSHMEM, we utilized the OSHMEM implementation included as part of the OpenMPI 4.0.1 package.

2) Performance Analysis: We first measure and compare the runtime statistics of the xBGAS and OpenSHMEM to study the performance impact of the xBGAS. As shown in Figure 7, xBGAS achieves impressive performance enhancements over the tested workloads. Particularly, the performance of the GATHER, GUPS (Giga-Updates Per Second), RANDPUT, and SCATTER benchmarks are improved by over 30%. On average, xBGAS boosts the performance of the tested workloads by 21.96% (up to 37.29%).

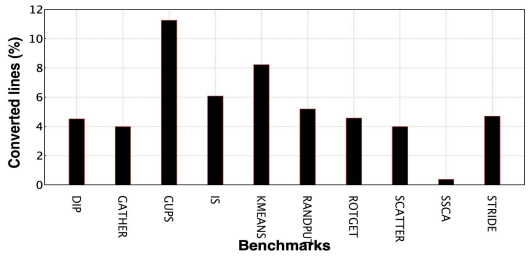

Fig. 13: Portability Analysis Data

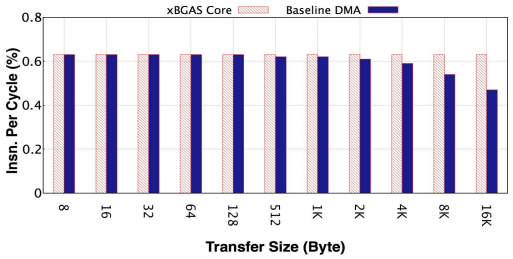

Fig. 14: Instruction Per Cycle

These performance enhancements can be attributed to the xBGAS micro-architecture support that significantly reduces the overhead of redundant software in communication. As such, we measure the proportion of network I/O latency and software overhead when conveying 1B∼1MB data trunks with xBGAS *get* and *put* operations. As shown in Figures 8 and 9, the trends of increasing network I/O latency proportion are observed as payload size increases for both the xBGAS *get* and *put* operations. Noticeably, xBGAS significantly reduces the software cost of loading a remote register-width data element to 9.72%, which implies a software overhead reduction of 78.43%, as compared to the OpenSHMEM *get* operations presented in Figure 1. Similarly, xBGAS *put* operations with small payloads only induce 9.70% of the software overhead during the inter-node communication, which further exhibits the performance impacts of xBGAS on irregular data-intensive workloads that produce small and sparse memory requests.

We also capture the executed instruction counts during the process of remote data operations with xBGAS and OpenSHMEM. We then derive the proportion of instructions reduced by xBGAS. The results are shown in Figure 10. We observe that xBGAS dramatically eliminates the software communication overhead in each test suite. Overall, xBGAS reduces redundant instruction execution by 69.26% on average, which effectively reduces the latency of inter-node communications.

3) Payload Analysis: As reported in Figure 11, we record the instruction counts of broadcasting distinct payload sizes between six nodes using the xBGAS and OpenSHMEM, respectively. Overall, the xBGAS model invokes fewer instructions (between 3.41x and 22.01x less) as compared to OpenSHMEM. Following the payload size increases from 8B to 32KB, the software cost of broadcasting in xBGAS remains more stable than that of OpenSHMEM, as the instruction count increases by 2x instead of 9.95x.

Further, each request/response message transferred via the NoC is broken up into one or more packets. Each message contains a 32-bit header with the message type and length. Using this protocol, each pair of request and response transactions require 8B of network control overhead in addition to the actual payload. As such, the request size can have great impacts on network resource utilization, e.g. bandwidth and buffer space. This behavior is observable in Figure 12, where we show the proportions of payload and control overhead, when employing different request granularities. Following the request size increases from 1B to 1KB, the percentage of bandwidth consumed by the payload grows from 11.11% to 99.22%. This observation implies that it will waste the majority of the bandwidth and network resources on network control overhead to dispatch small requests (1B∼8B) to accomplish large data transfer. Therefore, large requests require bulk transactions to effectively eliminate redundant network traffic and use bandwidth efficiently.

4) Portability Analysis: Furthermore, we also analyze the portability of the xBGAS. As the xBGAS API is both semantically and syntactically similar to the OpenSHMEM interface, the application porting procedure is greatly simplified. In order to quantify the degree of portability of xBGAS applications, we count the number of converted functions in lines of code and derive the percentage of modified lines in each test suite. As presented in Figure 13, only 4.99% lines of a program are modified on average to port benchmarks from the OpenSHMEM to xBGAS. These observations reveal a high degree of xBGAS program portability, which enhances the generalizability of the xBGAS solution.

B. RTL Based Simulation

1) Baseline Analysis: In order to evaluate the xBGAS against existing DMA and RDMA architectures, we compare the number of instructions executed, network protocol differences, and total transfer latency of different systems in Sections VI-B2, VI-B3 and VI-B4, respectively. The tested systems are: 1) a baseline DMA system, 2) an xBGAS system that does not leverage the bulk transfer interface, and 3) an xBGAS system that does leverage the bulk transfer interface.

The baseline DMA system is a Xilinx Zynq XC7Z020 SoC with two ARM Cortex-A9 processors clocked at 666.6MHz [45]. This baseline was chosen because no RISC-V based architecture with a DMA engine was available to the authors at the time of writing. ARM was chosen over other available architectures (including x86) because it was the only available RISC architecture with a DMA engine. As RISC-V and ARM are both RISC ISAs, their executed instruction counts will be similar for tasks without specialized instructions.

To ensure the baseline DMA system is similar to the xBGAS core, we compare the baseline DMA system's IPC with the average IPC of the xBGAS core presented in Table II. Figure 14 presents the results. Note that for transfers of up to 256B, the IPCs of the xBGAS core and baseline DMA system

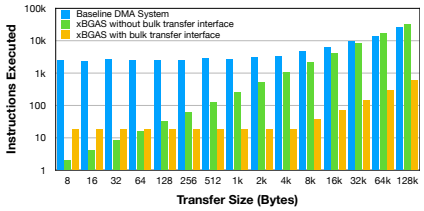

Fig. 15: Instruction Counts

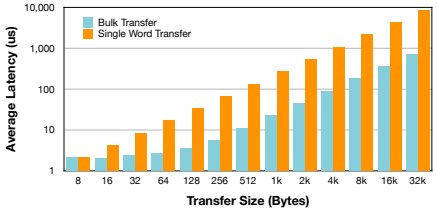

Fig. 16: NoC Latency

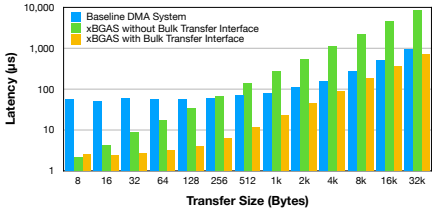

Fig. 17: Total Latency

are identical. The IPCs remain similar for transfers of up to 2KB. The IPC of the baseline system starts to drop as transfers exceed the 4KB DMA packet limit.

2) Instruction Analysis: In order to measure the number of executed instructions on the baseline DMA system, 10 million DMA transfers of each transfer size were executed with a "bare metal" C program. The average length of each transfer is calculated by recording the length of time needed to complete all DMA transfers. The system's IPC performance counter is recorded for the duration of the DMA transfers. The IPC, runtime of the test, and the system clock frequency are used to compute the number of instructions executed during the test. The number of instructions executed in both xBGAS systems (with and without the bulk transfer interface) are calculated statically based on the xBGAS runtime library assembly code. Figure 15 plots the instructions executed for each of the three systems at transfer sizes ranging from 8B to 128KB.

The baseline DMA system limits DMA packet sizes to 4KB and uses multiple packets for larger transfers. This packet size limitation is evident in the plot as the instructions executed per baseline DMA transfer stays flat for transfer sizes less than 4KB and grows linearly for transfer sizes of 4KB and larger. The instructions executed per xBGAS bulk transfer also grows linearly after 4KB because our static analysis of the xBGAS runtime library assumes the DMA engine behind the xBGAS bulk transfer interface has the same 4KB packet size limit.

Figure 15 shows that for transfers of 64B or less, the xBGAS single-element transfer interface requires the fewest executed instructions. For transfers greater than 64B, the xBGAS bulk transfer interface completes transfers with the fewest executed instructions. The baseline DMA system always requires more instructions to complete a transfer than the xBGAS bulk transfer interface. Moreover, even xBGAS transfers without the bulk transfer interface can complete transfers up to 32KB with fewer executed instructions than the baseline DMA system.

Although the xBGAS bulk transfer interface requires more instructions for small transfers, after a transfer is initialized, an xBGAS core is free to execute other instructions instead of waiting for issued global loads or stores to complete, albeit data dependencies are satisfied. On the other hand, if contention occurs in the bulk transfer, performing transfers without it may yield lower latency. The optimal transfer method is highly dependent on the transfer size and current system load. Providing the xBGAS runtime library with two techniques to complete transfers (the single-element or bulk transfer interface) allows it to select the best option based on the current system load and resource usage.

3) Network Protocol Analysis: In addition to lower instruction counts, bulk transfers with a regular DMA engine or xBGAS bulk transfer interface can provide reduced network latency because of their lower network overhead. If the xBGAS runtime library must use individual load and store instructions to execute a transfer, each instruction will be sent over the network as an individual message with all of the associated network control overhead. As shown in Figure 12, most of the data transmitted by small messages is network control overhead. Leveraging bulk transfers leads to larger message sizes, lower network control overheads, and more efficient network resource utilization.

We measure the latency of various network transfers to demonstrate the impact of bulk messages on network latency. Transfers initiated by the baseline DMA system or xBGAS bulk transfer interface perform transfers with one or more packets up to 4KB in size. Transfers initiated by xBGAS that do not leverage the bulk transfer interface transmit as many one word (4B) packets as needed to transfer a whole message.

The latency of transfers is measured with a NoC RTL simulation. The NoC is configured as an 8x8 mesh. The NoC routers are clocked at 75MHz. Each node in the mesh repeatedly performs transfers of a constant size to a random address. A simulation is performed for each transfer size. Each simulation runs for 10,000 cycles. The transfer latencies are recorded and averaged at the end of the simulation. Figure 16 plots the average latency for bulk packets and single word transfers. As expected, the larger messages with bulk transfers lead to lower latencies because of the lower network control overhead, resulting in more efficient network resource usage.

4) End-to-End Latency Analysis: In order to estimate transfer latency, the network transmission latency in Figure 16 is added to the computed instruction execution runtime based on instruction counts presented in Figure 15. Instruction counts are converted to runtimes based on the clock frequency and IPC of the xBGAS core presented in Table II. Estimating runtime with xBGAS IPC models the execution time of the baseline DMA transfers on an xBGAS system with a DMA engine or bulk transfer interface. A 75MHz clock is used to ensure the core operates below its Fmax reported in Table II. Figure 17 plots the total latency for different transfer sizes.

The xBGAS system without the bulk transfer interface provides the lowest latency transfers for 8B messages. However,

the overhead of single word NoC messages quickly catches up to the system. Without using the bulk transfer interface, xBGAS can provide lower latency transfers than the baseline DMA system for transfers up to 128B. Of the three systems, for transfers larger than 8B, the xBGAS bulk transfer interface provides the lowest latency transfers because of its efficient interface and use of large packets in the NoC.

VII. Conclusion

In this work, we have presented xBGAS, a novel RISC-V ISA extension providing a scalable global address space for HPC systems. The xBGAS maps discrete shared resources into an extended global address space (up to 128 bits) for efficient inter-component communication. We presented and analyzed a new ISA-level communication methodology, micro-architecture designs, and an associated programming model utilizing extended xBGAS instructions and registers. Further, we also designed the xBGAS runtime library as an interface to enhance the portability and programmability of xBGAS. As demonstrated by our evaluations, xBGAS reduces 69.26% of the instructions generated by communication calls and exhibits a 21.96% performance improvement on average, as compared to OpenSHMEM. These results and observations illustrate the potential impact of xBGAS on scalable HPC system design.

References

[1] Ed Anderson, Jeff Brooks, Charles Grassl, and Steve Scott. Performance of the Cray T3E multiprocessor. In *SC*. ACM, 1997.

[2] Yuichiro Ajima, Takahiro Kawashima, Takayuki Okamoto, Naoyuki Shida, Kouichi Hirai, Toshiyuki Shimizu, Shinya Hiramoto, Yoshiro Ikeda, Takahide Yoshikawa, Kenji Uchida, et al. The Tofu Interconnect D. In *CLUSTER*. IEEE, 2018.

[3] Jian Zhang, Chunbao Zhou, Yangang Wang, Lili Ju, Qiang Du, Xuebin Chi, Dongsheng Xu, Dexun Chen, Yong Liu, and Zhao Liu. Extreme-scale phase field simulations of coarsening dynamics on the sunway taihulight supercomputer. In *SC*. IEEE, 2016.

[4] Min Xie, Yutong Lu, Kefei Wang, Lu Liu, Hongjia Cao, et al. Tianhe-1a interconnect and message-passing services. *IEEE Micro*, 2011.

[5] John D Leidel, Xi Wang, Frank Conlon, Yong Chen, David Donofrio, Farzad Fatollahi-Fard, and Kurt Keville. xbgas: Toward a risc-v isa extension for global, scalable shared memory. In *MCHPC*, 2018.

[6] Jianbo Dong, Rui Hou, Michael Huang, Tao Jiang, Boyan Zhao, Sally A McKee, Haibin Wang, Xiaosong Cui, and Lixin Zhang. Venice: Exploring server architectures for effective resource sharing. In *HPCA*. IEEE, 2016.

[7] Guojing Cong, George Almasi, and Vijay Saraswat. Fast PGAS implementation of distributed graph algorithms. In *SC*. IEEE, 2010.

[8] Michail Alvanos, Montse Farreras, Ettore Tiotto, José Nelson Amaral, and Xavier Martorell. Improving communication in PGAS environments: Static and dynamic coalescing in UPC. In *ICS*. ACM, 2013.

[9] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. UPC++: a PGAS extension for C++. In *IPDPS*. IEEE, 2014.

[10] GenZ Consortium. http://genzconsortium.org. Accessed: 2018-11-28.

[11] CCIX Consortium. https://www.ccixconsortium.com. Accessed: 2019-1-10.

[12] OpenCAPI Consortium. https://opencapi.org. Accessed: 2019-4-7.

[13] xBGAS GNU Compiler Toolchain. https://github.com/tactcomplabs/xbgas-gnu-toolchain.

[14] xBGAS LLVM Compiler Toolchain. https://github.com/tactcomplabs/xbgas-llvm.

[15] xBGAS Assembly Test Suite. https://github.com/tactcomplabs/xbgas-asm-test.

[16] xBGAS Benchmarks. https://github.com/tactcomplabs/xbgas-bench.

[17] xBGAS Machine-Level Runtime Library. https://github.com/tactcomplabs/xbgas-runtime.

[18] xBGAS Simulation Toolchain. https://github.com/tactcomplabs/xbgas-tools.

[19] TOP500 List of Supercomputers. Technical report, November 2019. https://www.top500.org/lists/2019/11/.

[20] RV128 Specification. Technical report, 2018. https://github.com/riscv/riscv-isa-manual/blob/master/src/rv128.tex.

[21] Hung-Hsun Su, Max Billingsley, and Alan D George. Parallel performance wizard: A performance analysis tool for partitioned global-address-space programming. In *IPDPS*. IEEE, 2008.

[22] Scott Davidson, Shaolin Xie, Christopher Torng, Khalid Al-Hawai, Austin Rovinski, Tutu Ajayi, Luis Vega, Chun Zhao, Ritchie Zhao, Steve Dai, et al. The Celerity open-source 511-core RISC-V tiered accelerator fabric: Fast architectures and design methodologies for fast chips. *IEEE Micro*, 2018.

[23] Dai Cheol Jung, Scott Davidson, Chun Zhao, Dustin Richmond, and Michael Bedford Taylor. Ruche networks: Wire-maximal, no-fuss nocs: Special session paper. In *NOCS, 2020*.

[24] UCS Profiling Tool for OpenUCX. Technical report, February 2019.

[25] Sam Ainsworth and Timothy M Jones. Software prefetching for indirect memory accesses. In *CGO*. IEEE Press, 2017.

[26] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. Meet the walkers: Accelerating index traversals for in-memory databases. In *MICRO*. ACM, 2013.

[27] Snehasish Kumar, Arrvindh Shriraman, Vijayalakshmi Srinivasan, Dan Lin, and Jordon Phillips. SQRL: hardware accelerator for collecting software data structures. In *PACT*. ACM, 2014.

[28] RH Arpaci, DE Culler, A Krishnamurthy, SG Steinberg, and K Yelick. Empirical evaluation of the CRAY-T3D: a compiler perspective. In *ISCA*. IEEE, 1995.

[29] Vijay Karamcheti and Andrew A Chien. A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D. In *ISCA*. IEEE, 1995.

[30] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In *ASPLOS*. ACM, 2014.

[31] Steven L Scott. Synchronization and communication in the t3e multiprocessor. In *ACM SIGPLAN Notices*. ACM, 1996.

[32] Andrew Waterman, Yunsup Lee, David A Patterson, and Krste Asanovic. The RISC-V Instruction Set Manual, Volume I: Base User-level ISA. *UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, 116, 2011.

[33] Jonathan Balkind, Michael McKeown, Yaosheng Fu, Tri Nguyen, Yanqi Zhou, Alexey Lavrov, Mohammad Shahrad, Adi Fuchs, Samuel Payne, Xiaohua Liang, et al. Openpiton: An open source manycore research framework. In *ACM SIGARCH Computer Architecture News*, 2016.

[34] John D Leidel, Xi Wang, and Yong Chen. Goblincore-64: A risc-v based architecture for data intensive computing. In *HPEC*. IEEE, 2018.

[35] Andreas Traber, Florian Zaruba, Sven Stucki, Antonio Pullini, Germain Haugou, Eric Flamand, Frank K Gurkaynak, and Luca Benini. Pulpino: A small single-core risc-v soc. In *3rd RISCV Workshop*, 2016.

[36] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th {USENIX} Security Symposium ({USENIX} Security 16)*, pages 857–874, 2016.

[37] xBGAS Architecture Specification. https://github.com/tactcomplabs/xbgas-archspec, 2018.

[38] Xi Wang, Brody Williams, John D Leidel, Alan Ehret, Michel Kinsy, and Yong Chen. Remote Atomic Extension (RAE) for Scalable High Performance Computing. In *DAC*. IEEE, 2020.

[39] Arun F Rodrigues, K Scott Hemmert, Brian W Barrett, Chad Kersey, Ron Oldfield, Marlo Weston, Rolf Risen, Jeanine Cook, Paul Rosenfeld, E CooperBalls, et al. The structural simulation toolkit. *SIGMETRICS Performance Evaluation Review*, 38(4):37–42, 2011.

[40] Sahan Bandara, Alan Ehret, Donato Kava, and Michel Kinsy. Brisc-v: An open-source architecture design space exploration toolbox. In *FPGA'19*, New York, NY, USA, 2019. ACM.

[41] OpenSHMEM Example Test Suites. https://github.com/openshmem-org/openshmem-examples.

[42] Oak Ridge OpenSHMEM Benchmarks. https://github.com/ornl-languages/osb.

[43] David Bader and Kamesh Madduri. Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors. *HiPC 2005*, 3769:465–476, 2005.

[44] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon. NAS Parallel Benchmark Results. In *SC*. IEEE Computer Society Press, 1992.

[45] Xilinx. *Zynq-7000 SoC Technical Reference Manual*, July 2018.