Towards xBGAS on CHERI: Supporting a Secure Global Memory

Mert Side*, Brody Williams*, John Leidel†, Jonathan Woodruff‡, Simon W. Moore‡, Yong Chen*
*Texas Tech University, Lubbock, TX, USA
{mert.side, brody.williams, yong.chen}@ttu.edu
†Tactical Computing Labs, Muenster, TX, USA
jleidel@tactcomplabs.com
‡University of Cambridge, Cambridge, UK
{jonathan.woodruff, simon.moore}@cl.cam.ac.uk

***Abstract*—Modern computing systems embrace heterogeneous system architectures to meet high-performance computing requirements. As a result, the Extended Base Global Address Space (xBGAS) project introduced an extension to the RISC-V instruction set architecture (ISA) to provide extended addressing capabilities. Although the xBGAS project is currently implemented on the RISC-V ISA, the project's central vision is to include a broad domain of ideas that is not dependent on any single microarchitecture. Therefore, demonstrating the existing xBGAS benchmarks on the ARM ISA would enable working on a real system prototype. The ARM Morello platform is intended as an industrial demonstrator of a capability architecture using Capability Hardware Enhanced RISC Instructions (CHERI). Consequently, understanding the CHERI semantics and identifying the desired features for xBGAS are crucial for making xBGAS portable across platforms. In this work, we investigate creating a software-based xBGAS solution for the Morello platform by porting the xBGAS runtime onto the ARM ISA. First, we revisit the basics of the xBGAS project and the CHERI project to learn what the security benefits of CHERI on xBGAS are by taking advantage of CHERI capabilities. Then, we explore the implementation of the CHERI capabilities and Morello simulators to discuss an adaptation of the scalable xBGAS runtime library to support other ISAs such as ARM.**

***Index Terms*—xBGAS, RISC-V, CHERI, ARM, Morello, instruction set architecture, microarchitecture, shared memory**

I. Introduction

In the age of exascale computing, reducing latency on operations over remote devices is more critical than ever. High Performance Computing (HPC) is a field that finds itself at the crossroads for developing new advances in fabrication and communication technologies. Performance and speed are of the essence in the HPC environment. Each computing component, such as processors, accelerators, high-bandwidth memory, and network interface cards, has evolved individually to meet the needs of the time. The transition from monolithic architectures to scalable systems introduced challenges for supporting interactions across these components. Traditional approaches to solving these challenges introduce massive multi-layered software infrastructures to bridge each component. These approaches typically involve offering some parallel programming model in which the programmer is responsible for optimizing the overlap between the computation and communication. However, this multi-layered software paradigm in large-scale parallel applications often leads to higher complexity and performance degradation.

Partitioned Global Address Space (PGAS) [2] programming models are developed as an easy-to-use interface that simulates shared memory semantics across remote resources. Extended communication latencies associated with software overheads are a natural byproduct of this abstraction compounded by the increasing degree of heterogeneity inherent to HPC systems. The Extended Base Global Address Space (xBGAS) [5], [6] project offers an extended addressing model to the standard RISC-V instruction set architecture through the use of proposed microarchitecture extensions. xBGAS has been shown to increase significantly the performance of PGAS models in high-performance environments [9], which is one of the many application domains that could benefit from such an approach. If shared across heterogenous device types, xBGAS architecture could improve communication performance by hardware acceleration that was previously done only in software.

In parallel to this, hybrid capability architecture research has emerged to improve system security through the use of similar microarchitecture modifications. The Capability Hardware Enhanced RISC Instructions (CHERI) [7] is a hybrid capability architecture that blends architectural capabilities with hardware-supported descriptions of permissions on pointers. This allows traditionally memory-unsafe programming languages, such as C and C++, to be adapted for providing efficient protections. The success of CHERI has led to the development of the CHERI-enabled ARM Morello platform.

This work presents initial research into building a CHERI-enabled PGAS programming model as a step towards a more fundamental integration of the xBGAS and CHERI architectures. In this paper, we begin by revisiting the fundamentals of xBGAS, CHERI, and PGAS programming models and discuss our ongoing research for the vision to secure heterogeneous devices using the xBGAS extension.

II. Background

This study relies on numerous efforts on extended addressing capabilities and memory protection models to propose a novel software-based solution to bring these two worlds together. Bringing xBGAS to CHERI-enabled platforms will

combine the memory protections CHERI offers with the scalability of xBGAS. This section provides an overview of the xBGAS microarchitecture extension and the CHERI architecture.

A. xBGAS: Extended Base Global Address Space

To overcome the extended communication latencies that are typically associated with PGAS models, Leidel et al. [5] introduced initial research into accessing globally shared memory blocks by direct use of architectural instructions. This RISC-V microarchitecture extension, the Extended Base Global Address Space, provides global and scalable memory addressing support for high-performance computing architectures. A full-fledged platform incorporating these devices in a unified design is essential for widespread adoption. However, for the purpose of broader adoption and evaluations, a runtime API needed to be developed.

Williams et al. [9] asserted that there is some potential for improvement in the xBGAS microarchitecture extension. They present the xBGAS runtime API consisting of the initial implementation for collective communication operations. Moreover, they evaluated the performance and the feasibility of the xBGAS project by running simulations. Wang et al. [6] provided an overview of an architecture design where each processor for xBGAS is extended with two additional hardware components – i.e., the arbiter and the Namespace Lookaside Buffer (NLB). These additions assist in the elimination of software overheads that are traditionally associated with remote shared data accesses. Specifically, xBGAS has been shown to significantly increase the performance of PGAS models in high-performance environments. Overall, this paper emphasizes the significance of xBGAS in designing a scalable and generalizable HPC system. Nevertheless, the security of such systems is left as an open question.

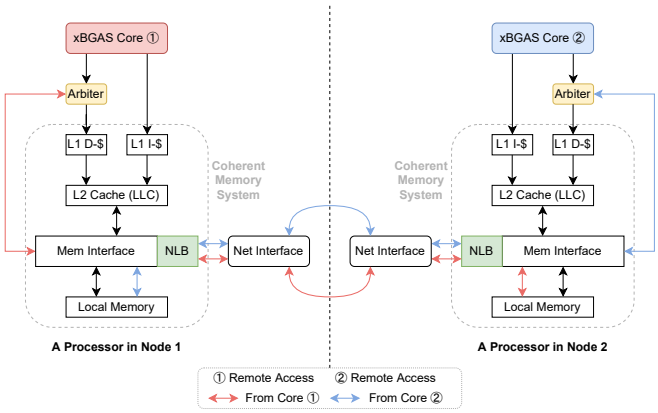

Fig. 1. The overview of xBGAS architecture design.

Figure 1 shows an overview of the xBGAS architecture design as used to facilitate remote memory accesses between nodes. Data requests are routed based on whether it requires a local or global operation via an arbiter directly attached to the CPU. Local accesses are handled on the local memory system via traditional means, while global accesses use extended addresses to be forwarded to remote nodes using the NLB. A mapping between recently used namespace IDs and remote node addresses is handled by the NLB that is contained within the memory interface of each processor. The NLB is a fully associative cache with a *least recently used* replacement policy.

In the xBGAS addressing model, where the upper 64-bits of the extended address serve as a namespace ID to specify the target remote node, and the lower 64-bit base address serves as a standard address in remote memory.

B. CHERI: Capability Hardware Enhanced RISC Instructions

Over the last decade, the University of Cambridge has extended conventional ISAs to enable memory protections and software compartmentalization [7]. These memory protections mitigate the software vulnerabilities derived from traditionally memory-unsafe programming languages such as C and C++ using fail-stop traps. Similarly, software compartmentalization limits malicious software's impact by decoupling the operating system and application code.

Similarly, Watson et al. [8] described how CHERI improves system security by utilizing both the hardware and software to bring higher compartmentalization performance and better granularity than the state-of-the-art capability architectures. For our research, we are interested in combining the security benefits of this architecture with the scalable addressing model proposed by xBGAS.

To implement capabilities, CHERI defines an additional set of registers. Special capability instructions are used to access the capability register file. This allows *load* and *store* of capabilities from memory for inspection/manipulation for dereferencing and for targeting from jump and branch instructions [10]. Accessing a capability starts by checking its validity tag, relocation relative to its base and offset, bounds checking relative to its base and length, and permission checking [8]. Operations allowed to be performed on a capability are determined by capability permissions control. Figure 2 depicts the CHERI capability representation where each capability consists of a 64-bit address and some additional metadata that is compressed to fit in the remaining 64 bits of the capability.

Fig. 2. The CHERI capability representation with 64-bit address plus metadata in addressable memory along with 1-bit validity tag

The research on the CHERI protection model by the University of Cambridge presented a unique architectural extension that led the ARM semiconductor company to start a research program named the ARM Morello program [3]. Considering the interest in capability-based addressing for adopting new security behavior in software, Bauereiss et al. [1] constructed a formal model of Morello, and Grisenthwaite et al [3] prototyped a CHERI-enabled architecture and board by extending the existing ARM Neoverse N1.

III. Methodology

The CHERI architecture generates a number of interesting possibilities for the integration of xBGAS. To test the feasibility of our design, we developed a simplified xBGAS (OpenSHMEM-like) runtime. Layering a special-purpose PGAS API on top of the novel Morello platform enables us to work with CHERI's architectural features.

A. Overview

Porting the xBGAS runtime library onto the CHERI-enabled Morello boards consists of work in four main abstraction layers shown in Figure 3. First, to emulate a real Morello SoC, we utilize ARM's Fixed-Virtual Platform (FVP) ecosystem. The FVP provides a *functionality-accurate* programmer's view of the hardware platform using the binary translation technology running at speeds comparable to the real hardware. Second, we modify the low-level runtime by translating the xBGAS API assembly functions written in RISC-V ISA to ARM ISA. Third, we leverage thread pooling on the high-level runtime to model the behavior of multiple processes. Finally, we propose minimal changes to the existing xBGAS benchmarks by only requiring the developer to add an entry point and an exit point in their programs to define the code segment that multiple threads can execute in parallel.

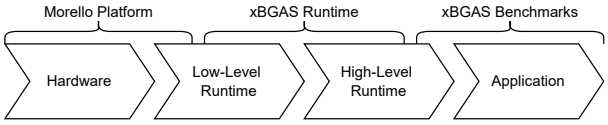

Fig. 3. The overview of porting the xBGAS runtime.

As our simulation platform, we use ARM's Fixed Virtual Platforms (FVPs). The Morello FVP uses binary translation technology to mimic hardware subsystems. It provides functional simulations of essential hardware components such as processors, memory, and peripherals. From the programmer's perspective, the FVP enables the execution of the full software stack offered by the SoC.

B. Challenges

Since the CHERI capabilities are realized within the scope of virtual address space, passing data via interprocess communication (IPC) would break the validity of capabilities. Therefore, realizing shared capabilities (i.e., sharing memory across processes) is the first challenge to be addressed. The xBGAS introduces additional hardware, such as an arbiter and an NLB, to handle local and remote operations. Similarly, a CHERI+PGAS implementation must administer a method to encode data location. Hence, working across distributed memory is another challenge. A solution needs to be devised on Morello so that the *remoteness* of data can be communicated.

We start our efforts by utilizing CHERI-enabled microarchitecture extensions when possible and introducing software abstractions where needed. One approach for realizing shared capabilities could be to use an mmap-based solution via `fork`. It is uncertain whether this would violate the integrity of capabilities from a security perspective. Another approach would be to shared address space across launched processes using the UNIX co-processes. However, the current implementation of the UNIX co-process model on the Morello boards is highly experimental; therefore, we focus on modeling our runtime using POSIX threads (`pthreads`).

Regarding hardware requirements, xBGAS's need for an arbiter can be simulated via a software abstraction. However, the NLB functionality requires some bits on the memory interface to encode data location. We propose using the capability object field in CHERI to encode *remoteness* information. Sacrificing 15 bits from this field would offer 32768 unique IDs for remote nodes, which would be sufficient at node-level prototyping.

C. Thread Pooling

Influence by the heterogeneous approach to high performance runtime techniques outlined by Leidel et al. [4], we use thread pooling to address the aforementioned challenges. Since IPC breaks the validity of capabilities and the xBGAS relies on shared memory across processes, we take advantage of thread pooling. This also enables us to keep the runtime lightweight by reducing the overheads associated with communication and thread creation.

A *Thread Pool* is a software design paradigm that is typically used for achieving concurrency in a computer program using multiple threads. Since the creation of threads comes with some overhead, managing a large number of threads can become inefficient. Using a pool of threads that efficiently delegates incoming jobs simplifies thread management and overcomes this overhead. In general, thread pools are implemented by creating a fixed number of threads at the program's initialization. As the program runs, small chunks of tasks are assigned to each thread for execution. When threads are done processing their tasks, they are marked as completed. The idling threads then pull a new task from a queue that is waiting to be processed.

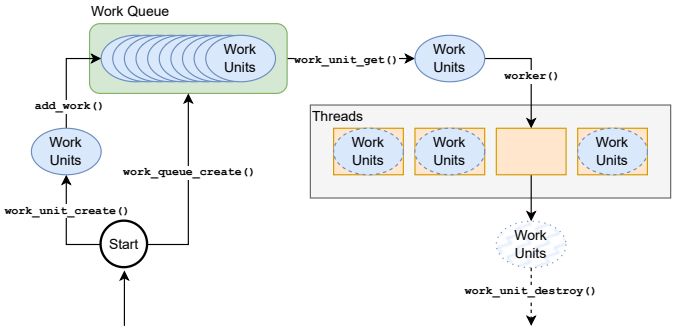

Fig. 4. The Flowchart of thread management.

The *Work Units* are small units of task that can are fed into a thread. A *Work Unit* contains three members, all of which are 64-bit pointers. The first pointer is to store the address of the function to be executed via a thread. The second pointer is to handle the arguments to be passed on to that function.

The third pointer points to the next *Work Unit* to get executed subsequently, similar to a singly linked list. Each *Work Unit* waits to be mapped onto a thread to get processed.

The *Work Queue* is a singly linked list that is designed to assist the management of *Work Units*. A *Work Queue* contains eight members. Two members are used as *Work Unit* pointers to mark the head and the tail of the queue. Another two members mark `pthread` conditions to signal work status. A member marks the mutex for all necessary locking. Two members are unsigned integers to track the number of threads that are alive and the number of threads that are actively working. Finally, the last member is a flag to tell the threads to stop before the destruction of the *Work Queue*.

D. The xBGAS API

On the xBGAS API, remote data communication is accomplished through one-sided remote get and put calls between processing elements. Therefore, we evaluate the feasibility of our porting effort by testing the sequential get and put calls between different threads. These functions make use of the xBGAS extended load and store assembly instructions. The prototype of the get function is shown below:

```
void xbrtime_TYPENAME_get(TYPE *dest,
const TYPE *src, size_t nelems, int
stride, int pe)
```

In these functions, `dest` expects a pointer to a destination address, `src` expects a pointer to a source address, `nelems` contains the total number of elements to be transferred, `stride` dictates the bytes between consecutive elements at `src` and `dest`, and `pe` contains the unique ID of the target processing element. Although not shown, non-blocking forms of both get and put are also included in the library for the given data types. As a further optimization, the underlying assembly code unrolls the loops when `nelems` exceeds a desired threshold.

Assembly 1 RISC-V

```
1:  __xbrtime_get_u8_seq:
2:      eaddie e10, a2, 0
3:      mv x31, zero
4:  .get_u8_seq:
5:      eld x30, 0(a0)
6:      add a0, a0, a4
7:      add x31,x31,1
8:      sd x30, 0(a1)
9:      add a1, a1, a4
10:     bne x31,a3,.get_u8_seq
11:     ret
```

Assembly 2 ARM Morello

```
1:  __xbrtime_get_u8_seq:
2:      MOV X12, XZR
3:  .get_u8_seq:
4:      LDR X10, [X0]
5:      ADD X0, X0, X3
6:      ADD X12, X12, #1
7:      STR X10, [X1]
8:      ADD X1, X1, X3
9:      CMP X12, X2
10:     BNE .get_u8_seq
11:     RET
```

As the existing xBGAS runtime library includes several sections of RISC-V assembly code to perform extended addressing, we translated these code segments to CHERI-compliant ARM assembly analogs in order to run on Morello. The code snippet above shows a simplified translation of an xBGAS Get call from RISC-V in Assembly 1 to ARM Morello in Assembly 2. Note that, since the xBGAS extended address management instruction on line 2 of Assembly 1 relies on architectural features not included in Morello, we omit a direct translation and instead simulate the desired behavior.

IV. Conclusion & Ongoing Work

This work presents the initial research behind building a CHERI-enabled PGAS programming model as a step towards a more fundamental integration of the xBGAS and CHERI architectures. The integration of these two architectures will provide secure and scalable systems with minimal performance overhead. True to the original goal of generalizability for xBGAS, this work also offers a roadmap toward bringing the xBGAS microarchitecture extension to the ARM ISA.

Although this work does not provide performance results on the xBGAS infrastructure, we have presented the methodology for achieving a lightweight runtime on CHERI platforms. Moreover, we have detailed our prototyping platform to be ARM's FVP for the Morello boards. The evaluations will likely be performed on the FVP using the xBGAS runtime's initialization, allocation, and transfer tests which will give a functionality-based simulation. Along with the functionality test described above, we also seek to evaluate our runtime on ARM Morello SoC hardware. This hardware is currently under testing and will be used to evaluate the feasibility and performance of xBGAS on CHERI.

References

[1] T. Bauereiss, B. Campbell, T. Sewell, A. Armstrong, L. Esswood, I. Stark, G. Barnes, R. N. Watson, and P. Sewell, "Verified Security for the Morello Capability-enhanced Prototype Arm Architecture," in *European Symposium on Programming*. Springer, Cham, 2022, pp. 174–203.

[2] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. Von Eicken, and K. Yelick, "Parallel programming in Split-C," in *Supercomputing'93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*. IEEE, 1993, pp. 262–273.

[3] R. Grisenthwaite, "Arm morello evaluation platform-validating cheri-based security in a high-performance system," in *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE Computer Society, 2022, pp. 1–22.

[4] J. D. Leidel, J. Bolding, and G. Rogers, "Toward a scalable heterogeneous runtime system for the convey mx architecture," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 2013, pp. 1597–1606.

[5] J. D. Leidel, X. Wang, F. Conlon, Y. Chen, D. Donofrio, F. Fatollahi-Fard, and K. Keville, "xBGAS: Toward a RISC-V ISA Extension for Global, Scalable Shared Memory," in *Proceedings of the Workshop on Memory Centric High Performance Computing*, 2018, pp. 22–26.

[6] X. Wang, J. D. Leidel, B. Williams, A. Ehret, M. Mark, M. A. Kinsy, and Y. Chen, "xBGAS: A Global Address Space Extension on RISC-V for High Performance Computing," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 454–463.

[7] R. N. M. Watson, S. W. Moore, P. Sewell, and P. Neumann. (2022, Jan) Capability Hardware Enhanced RISC Instructions (CHERI). [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/

[8] R. N. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie *et al.*, "Cheri: A hybrid capability-system architecture for scalable software compartmentalization," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 20–37.

[9] B. Williams, X. Wang, J. D. Leidel, and Y. Chen, "Collective communication for the RISC-V xBGAS ISA extension," in *Proceedings of the 48th International Conference on Parallel Processing: Workshops*, 2019, pp. 1–10.

[10] J. Woodruff, R. N. Watson, D. Chisnall, S. W. Moore, J. Anderson, B. Davis, B. Laurie, P. G. Neumann, R. Norton, and M. Roe, "The cheri capability model: Revisiting risc in an age of risk," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. IEEE, 2014, pp. 457–468.