GoblinCore-64: A RISC-V Based Architecture for Data Intensive Computing

John D. Leidel
Tactical Computing Labs &
Texas Tech University
Muenster, Texas, US
jleidel@tactcomplabs.com

Xi Wang
Department of Computer Science
Texas Tech Univeristy
Lubbock, Texas, US
xi.wang@ttu.edu

Yong Chen
Department of Computer Science
Texas Tech University
Lubbock, Texas, US
yong.chen@ttu.edu

***Abstract*—Current microprocessor architectures rely upon multi-level data caches and low degrees of concurrency to solve a wide range of applications. These architectures are well suited to efficiently executing applications that support memory access patterns with spatial and/or temporal locality. However, data intensive applications often access memory in an irregular manner that prevents optimal use of the memory hierarchy. In this work, we introduce GoblinCore-64 (GC64), a novel architecture that supports large-scale data intensive high performance computing workloads using a unique memory hierarchy coupled to a latency-hiding micro architecture.**

The GC64 infrastructure is a hierarchical set of modules designed to support concurrency and latency hiding. The memory hierarchy is constructed using an on-chip scratchpad and Hybrid Memory Cube 3D memories. The RISC-V based instruction set includes support for scatter/gather memory operations, task concurrency and task management. We demonstrate GC64 using standard benchmarks that include NAS, HPCG, BOTS and the GAP Benchmark Suite. We find that GC64 accelerates these workloads by up to 14X per core and improves bandwidth by 3.5X.

***Index Terms*—component, formatting, style, styling, insert**

I. Introduction

The current state of the art in mainstream microprocessor architectures continue to rely upon memory systems that consist of multi-level data caches and traditional DDR main memory devices. The native hardware concurrency mechanisms present in the respective micro-architectural implementations only provide a low degree of hardware managed concurrency. Further, these mechanisms are often difficult or entirely not visible from the application layer or instruction set architecture. These mechanisms often promote efficient utilization or near-optimal performance for applications with significant memory reuse or linear memory access patterns.

Contrary to this, applications that are generally considered to be data intensive may not make efficient use of the data caches present. These applications may access memory in random or highly irregular patterns, thus exhibiting cache misses on a high percentage of requests. These applications may also have very large working sets in main memory. The result of utilizing such large data sets is a significant probability that subsequent requests will not fall in existing hierarchical data caches.

This work introduces the GoblinCore-64 architecture and infrastructure [1]. GoblinCore-64, herein also referred to as *GC64*, combines a hierarchical system infrastructure and memory infrastructure to provide a scalable architecture designed to efficiently support data intensive computing applications. The system infrastructure hierarchy provides efficient concurrency mechanisms to hide the latency of accessing memory in irregular patterns. The memory infrastructure combines software-managed scratchpad memories on chip to high bandwidth, Hybrid Memory Cube devices in order to provide significant bandwidth to concurrent applications. The core machine model and instruction set are based upon the RISC-V instruction set architecture with the addition of an extended set of instructions designed to support task concurrency, task management and scatter/gather memory operations. Finally, every attempt has been made to build the hardware modules, infrastructure and software tools under BSD-like licenses such that both academic and commercial organizations may make use of GoblinCore-64.

The remainder of this work is organized as follows. Section II discusses previous attempts at constructing architectures for data intensive computing. Section III provides an overview of the machine organization and the RISC-V instruction set extensions. Sections IV and V provide our evaluation methodology and the associated results, respectively. We conclude with a summary of our results in Section VI.

II. Previous Work

Several previous architectures have made attempts to directly couple the hardware and software runtime architecture in order to efficiently support data intensive style applications. One of the original attempts was the Tera MTA [2] [3]. The MTA combined hardware-driven multithreading and tag-bit semantics in order to provide a whole-machine programming model [4] [5] [6]. The barrel processors present in the MTA support 128 threads per CPU and switch threads on every cycle.

Later revisions of the MTA were released as the Cray XMT and XMT-2 [7]. The XMT processors, codenamed *Threadstorm*, shared a socket design with the AMD Socket F processor and a system architecture shared with the Cray XT4.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

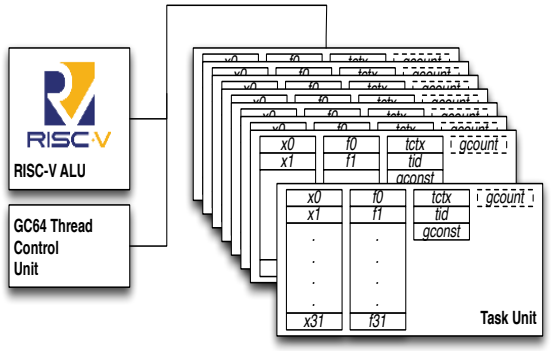

Fig. 1. GC64 Task Processor

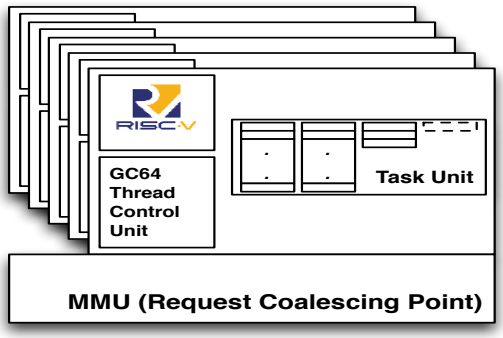

Fig. 2. GC64 Task Group

However, both the MTA and the XMT shared non-standard programming models that often hindered general acceptance.

The Convey MX-100 [8] [9] was also designed to execute data intensive applications using latency hiding techniques. The MX-100 was designed using a shared memory, heterogeneous system architecture that combined dual-socket Intel x86_64 servers to the MX100 coprocessor board via PCIe x8 Gen3 links. The MX-100 supported a simple, OpenMP-based threading and tasking model that was directly assisted by hardware concurrency mechanisms. The major downside to the architecture was the implementation of the coprocessor via low-frequency (300Mhz) FPGA logic. Despite its simple programming model, the result was an architecture that required significant application concurrency in order to garner performance.

The IBM Cyclops64, also known as the Blue Gene/C, provided an interesting memory hierarchy that consisted of global interleaved memories connected via a crossbar switch and local scratchpad memories [10][11]. The core architecture contained 80 processor per socket executing at 500Mhz. The Cyclops64 exposed much of the underlying machine architecture in order to provide programmers and the runtime stack sufficient access hardware in order to maximize performance and efficiency. One of the core programming models for the Cyclops64 was an OpenMP [12] implementation that exploited the hardware mechanisms present.

III. Machine Organization

The GoblinCore-64 machine hierarchy can be described in terms of three major units: the execution hierarchy, the memory hierarchy and the core architecture. The core architecture implements the core instruction set and concurrency instruction set extensions required to operate a single execution unit. The execution hierarchy implements a set of configurable, nested hardware mechanisms that define the breadth and depth of the available concurrency in the system. Finally, the memory hierarchy implements a set of primitive hardware units in conjunction with the execution hierarchy in order to transparently promote efficient use of memory bandwidth resources.

A. Execution Hierarchy

The execution hierarchy of GoblinCore-64 is designed to facilitate multiple designs and implementations with varying degrees of hardware concurrency and system scalability. We provide this functionality via a set of six hierarchical layers in the execution module hierarchy. Layers 0-3 are required of all devices that implement the GC64 architecture. Layers 4-5 are optional module layers that provide additional scalability beyond a single system-on-chip. Each subsequent layer requires all levels below it to ensure a correct implementation. The execution hierarchy layers are labeled as follows:

- Layer 0: Task Unit
- Layer 1: Task Processor
- Layer 2: Task Group
- Layer 3: Socket
- Layer 4: Node
- Layer 5: Partition

The GC64 task unit is the smallest divisible unit of concurrency. The task unit contains a single RISC-V integer register file, the optional RISC-V floating point register file, the GC64 user-visible registers and the GC64 machine state registers. The goal of the task unit is to provide a simple hardware mechanism that maps directly to a single unit of parallelism in software. Software runtime libraries have the ability to map constructs such as threads or tasks directly to a single task unit. We map one or more task units to a task processor. The GC64 task processor consists of the integer and optional floating point arithmetic units, a thread control unit as well as the associated task units (maximum of 256).

The task processor, as shown in Fig 1, is permitted to execute instructions from a single task unit on any given cycle. In this manner, a single task processor maintains control of the target core from a single task unit at any given time. It is the job of the thread control unit to enforce which task unit is in *focus* on any given cycle. Any time the thread control unit switches focus from one task unit to an adjacent task unit, we refer to this event as a *context switch*. The thread control unit may select a new task unit only from the task units that are directly attached. GC64 requires that at least one task unit exists per task processor. GC64 permits

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

a maximum of 256 task units per task processor. The thread control unit architecture and performance characteristics are further described in [13].

The GC64 Task Group, as shown in Fig 2, consists of one or more task processors interconnected to a local memory management unit (MMU) and subsequently to the SoC peripheral interfaces. This MMU serves two purposes. First, it determines whether a request is designated as a local request or a global request. Local requests are serviced by either the on-chip scratchpad memory or one or more HMC interfaces. Global requests are serviced by off-chip resources. Second, the local MMU performs request coalescing. Memory requests from multiple tasks across multiple task units and task processors are coalesced into larger request packets in order to optimize channel bandwidth utilization. The memory hierarchy is discussed further in Section III-B.

The GC64 socket architecture shown in Fig 3 combines multiple task groups (and their hierarchical components) into a system-on-chip package. This package has the ability to host up to 256 GC64 task groups under the current architecture specification. Individual task groups and periphery modules are interconnected via the OpenSoC Fabric network on chip infrastructure [14]. In addition to task groups, the GC64 socket contains a number of peripheral modules. First, it contains a scratchpad memory module that acts as a low-latency, high bandwidth, user-allocatable memory resource for commonly used data. It also contains an atomic memory operation (AMO) unit that controls queuing, ordering and arbitration of atomic memory operations. All of-chip messages or data are handled via one or more Hybrid Memory Cube [15] channel interfaces and/or the GC64 off-chip network interface.

The remaining two modules in the GC64 execution hierarchy are optional based upon the desire to scale the infrastructure out beyond a single socket. The node hierarchy consists of one or more GC64 socket modules that reside within a single node addressing domain. The architecture specification defines a maximum of 256 sockets per node and a maximum of 65,536 nodes per partition. Each GC64 partition consists of one or more nodes that share a top-level addressing domain. Partitions may also contain links to adjacent partitions in order to create much larger system configurations. The node and partitioning schema are architected to provide a natural boundary for defining memory locality domains in higher-level programming models while maintaining globally shared memory addressing. Further information regarding the scalability of the GC64 infrastructure is described in [16].

The maximum configured GC64 system architecture can be described as follows:

- 256 Task Units per Task Processor
- 256 Task Processors per Task Group
- 256 Task Groups per Socket
- 256 Sockets per Node
- 65,536 Nodes per Partition
- 65,536 Partitions per System
- Maximum task concurrency of 18,446,744,073,709,551,616 tasks

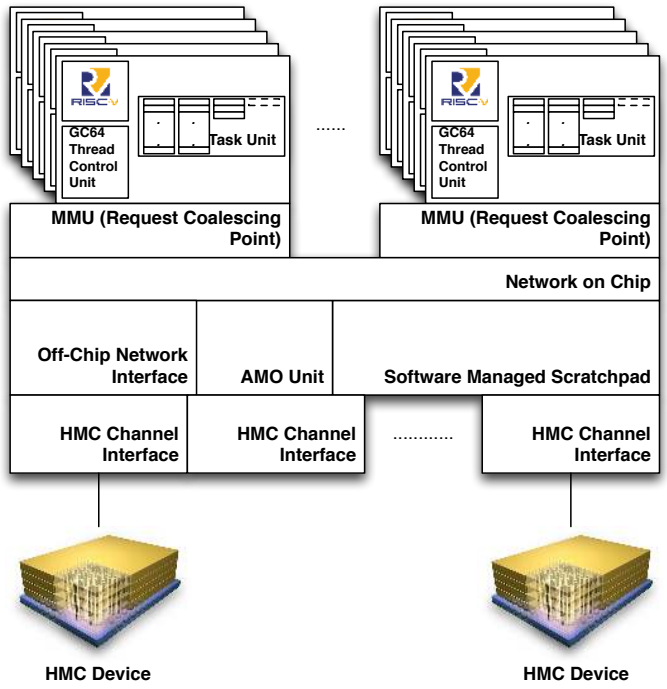

Fig. 3. GC64 Socket Architecture

B. Memory Hierarchy

The GC64 memory hierarchy is designed to promote efficient utilization of on-chip and system bandwidth resources for highly concurrent applications. As previously stated, these applications generally exceed the pathological memory traffic and bandwidth considerations for traditional system architectures. The memory hierarchy consists of three main components, including the memory management unit (MMU), the software-managed scratchpad memory and one or more Hybrid Memory Cube (HMC) devices.

The task group MMU serves two main purposes. First, it provides a memory coalescing point for all tasks in the respective task group. The coalescing logic is designed to build larger request payloads, where applicable, in order to service multiple individual memory requests with a single HMC request [17]. This functionality is a key differentiator in the GC64 infrastructure as we coalesce memory requests across tasks and task groups. In this manner, we exploit the high degree of concurrency in order to advantageously find spatial and temporal locality in the memory request patterns. As a result, applications that exhibit irregular memory accesses may still make advantageous use of the HMC memory bandwidth. The MMU also determines whether an individual memory request is designated as *local* or *global*. Local requests are those requests that are serviced by any HMC device or software managed scratchpad that is directly attached to the respective socket. Global requests are those that must be routed off-socket and thus serviced by adjacent sockets, nodes or partitions.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

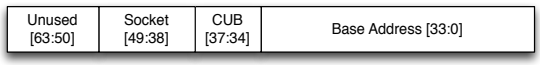

Fig. 4. GC64 Physical Address Format

The software managed scratchpad is the second component in the mandatory memory hierarchy and serves a purpose similar to a traditional data cache in that it contains frequently used data items that can be read or written to via any of the tasks in an application's process space. The final unit in the memory hierarchy is the HMC device [15]. The HMC devices may be interconnected directly or via routed devices (devices connected via other HMC devices). The GC64 micro-architecture does not prohibit any standard HMC configurations.

The (local) physical addressing format shown in Fig 4 is designed to support locally scalable (within a node) memory addressing the directly maps to the vault-interleaved nature of the HMC devices. The local physical memory address format consists of four fields as follows. The base address is a 34-bit physical address. The CUB, or *Cube ID*, field maps to the HMC request payload's definition of a Cube ID. CUB *0xf* maps to the local scratchpad memory. The Socket ID is an 8-bit field that identifies the socket ID within a GC64 node. Manipulating this field will initiate memory requests to adjacent sockets. Bits 50-63 are currently ignored and may be utilized for future expansion.

C. ISA Extension

The GoblinCore-64 core architecture consists of an implementation of the RISC-V instruction set architecture [18] as well as extensions specifically required to support the concurrency mechanisms required for GC64. The RISC-V instruction set architecture was specifically chosen due to its ground-up design to be an open, modular and extensible instruction set specification [19].

The GC64 core architecture defines a minimum number of RISC-V architectural extensions in order to support the target data intensive application profiles as well as the additional GC64 ISA extension. The GC64 core architecture requires the RV64I, M and A extensions in order to provide addressing and mutable memory state. In addition, GC64 supports optionally including the F and D extensions to enable hardware floating-point support and the RV128I for extended addressing support.

In addition to the base set of RISC-V extensions required to construct the GC64 core architecture, GC64 defines additional machine register state designed to support the required concurrency mechanisms in the micro architecture. The additional state includes eight registers, six user registers (U), one supervisor (S) and one arithmetic machine state (AMS) register (Table I). The additional register state cannot be directly accessed from the base RISC-V ISA or standard set of extensions. They can, however, be accessed from GC64 extended instructions.

TABLE I
GC64 REGISTER EXTENSIONS

Mnemonic	Index	Mode	Function
tctx	0b00000	U, S	Task address
tid	0b00001	U, S	Task ID
tq	0b00010	U, S	Task queue
te	0b00011	U, S	Task exception queue
gconst	0b00100	U, S	GC64 HART
garch	0b00101	U, S	GC64 architecture
gkey	0b10000	S	Process space key
gcount	N/A	AMS	Context switch task pressure

The *tctx* register provides the base address of the context save/restore region for each respective task defined using a unique task ID in *tid*. The *tq* and *te* registers store the address of the task queue and task exception queue, respectively. The task queue contains a doubly linked list of tasks ready to execute and the exception queue contains tasks in an exception (context save) state. The *gconst* register contains architecture-specific values for the physical locality of the executing task (task unit, task group, socket, etc) while the *garch* register contains configuration values describing the architectural dimensions of the GC64 SoC. The *gkey* register contains a kernel-initiated security key in order to prevent rogue processes from utilizing resources outside of its permissible process domain. Finally, the *gcount* register contains the pressure by which the associated task is exerting on the respective task processor [13].

In addition to the aforementioned register state, GC64 includes three additional classes of instructions that manipulate various portions of the extended infrastructure. The first class, or *concurrency instructions*, includes two instructions (*iwait* and *ctxsw*) that permit user applications to control the task concurrency mechanisms. The *iwait* instruction permits users to hold a task unit in the core for a specified number of cycles and thus override the GC64 context switch pressure policy. Conversely, the *ctxsw* instruction forces an immediate context switch, analogous in kind to an OpenMP task yield operation. Next, the *task control instructions* provide users the ability to spawn and join GC64 hardware tasks. The goal of these instructions is to provide the least possible latent mechanism by which to grow the concurrency of an executing application. Finally, the *scatter/gather instructions* add indexed load and store operations for all base integer and (optionally) floating point types.

IV. EVALUATION

A. Workloads

For our evaluation, we utilize a mixture of application workloads and benchmarks the are historically known to make use of dense memory operations and sparse memory operations. These representative applications also span a variety of traditional scientific (physical) simulation, biological simulation, core numerical solvers and analytics workloads. All of the workloads are compiled with optimization using

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

the RISC-V GCC 5.3.0 toolchain. We classify these workloads into four categories, *BOTS*, *GAPBS*, *NASPB* and *MISC*.

The first set of workloads is built using the Barcelona OpenMP Tasks Suite, or BOTS [20]. BOTS utilizes the tasking features from the OpenMP [21] shared memory programming model to demonstrate a series of pathological workloads from a variety of different scientific disciplines. The second set of workloads is provided by the GAP Benchmark Suite [22]. This suite of benchmarks is designed to provide a standardized set of workloads written in C++11 and OpenMP to evaluate a target platform's efficacy on large scale graph processing. The third set of workloads is provided by the NAS Parallel Benchmarks, or NASPB [23]. The OpenMP-C version of the benchmark source is utilized for our tests. We make use of the *A* problem size across all kernels and pseudo-applications. The final set of workloads include the HPCG benchmark [24] [25], LULESH [26] [27] and STREAM [28].

B. Methodology

For our simulation infrastructure, we utilize a heavily modified version of the RISC-V *Spike* simulator. As shown in Fig 5, our modifications encapsulate three major areas of the infrastructure. First, we added the necessary ISA extensions and internal logic in order to encapsulate the GC64 specification. This included register definitions, instructions and arithmetic machine state. Next, we modified the core timing model in order to support our context switch mechanisms [13] in a more cycle accurate manner. This included representing individual instructions by their relative cost in the Rocket 5-stage pipeline. Finally, we construct a series of external interfaces using named pipes in order to provide high fidelity external tracing mechanisms. These external tracing mechanisms provide the ability to track the efficacy of the internal GC64 components without significantly degrading the simulation performance or adding unnecessary internal state. For this study, we couple the GC64 context switch mechanisms and the memory coalescing mechanisms to external tracing applications in order to record timing, performance and utilization.

The simulation environment was executed using the RISC-V Linux kernel version 4.1.17 executing in SMP mode. In this manner, all system calls, I/O traffic and periphery application perturbations are captured in our memory coalescing and context switching results. For each of the aforementioned applications, we utilized the GCC 5.3.0 compiler with the standard GOMP OpenMP library implementation for thread and task parallelism.

Using this configuration, we split our evaluations into three sections. First, we evaluated the efficacy of the memory coalescing approach by executing each of the workloads on our simulation infrastructure with context switching disabled and coalescing enabled. We execute each workload using 1 to 32 threads in pure SMP mode and record the read, write and total number of memory requests that we coalesce into larger HMC payloads. Next, we execute the same series of workloads with both context switching enabled and memory

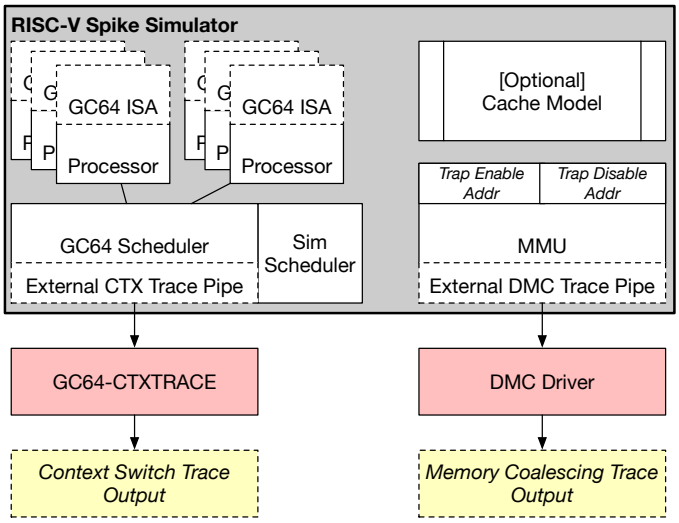

Fig. 5. Simulator External Trace Mechanisms

coalescing disabled using 2 and 8 task units per core and a maximum context switch pressure of 64 and 128 (four total configurations). These context switch benchmarks were executed using multiple tasks for a single core. Finally, we enable the memory coalescing and the context switching mechanisms in the simulator in order to record comprehensive performance results.

V. Results

As we see in Fig 6, nearly all the representative applications realize a significant reduction in the overall number of memory requests dispatched through to the memory controller. Across all applications and thread counts, we see an average memory request reduction of 49.74%. This implies that our proposed memory coalescing approach within each task group coalesces roughly half of the incoming read and write requests, which are subsequently translated to larger HMC packet requests. Given the packetized nature of the HMC device, each coalesced request requires lower relative control overhead per the respective data payload, thus permitting more efficient use of memory bandwidth.

Next, in Fig 7 we find the relative speedup obtained by utilizing the GC64 context switching mechanisms. The best workload represented in the graph is the BOTS Fibonacci benchmark which exhibited a 14.6X speedup over the baseline. Given the inherently recursive nature of Fibonacci, the context switching mechanisms make advantageous use of the GC64 resources despite the significant number of function calls. The next highest speedup was observed with the NASPB LU benchmark (5.53X). Conversely, the lowest speedup observed was the BOTS Health benchmark with a maximum speedup of 1.00X. Overall, we observe an average speedup across all application workloads of 3.23X.

From our context switch analysis, we may also observe that at least one parallel implementation for each workload exceeded the baseline configuration. As a result, we may

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

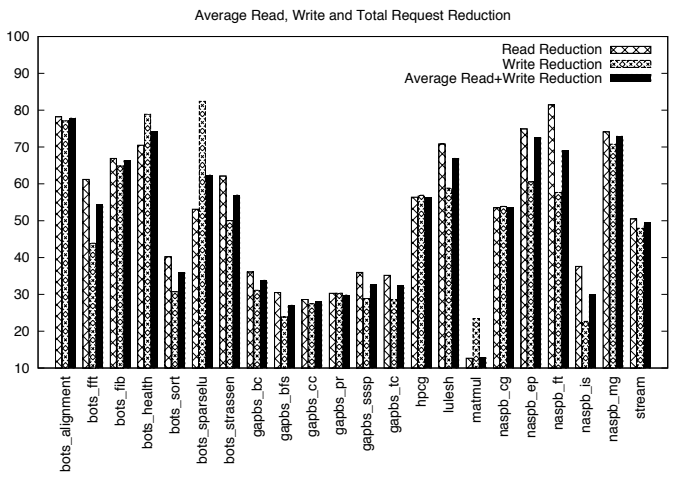

Fig. 6. Average Request Reduction

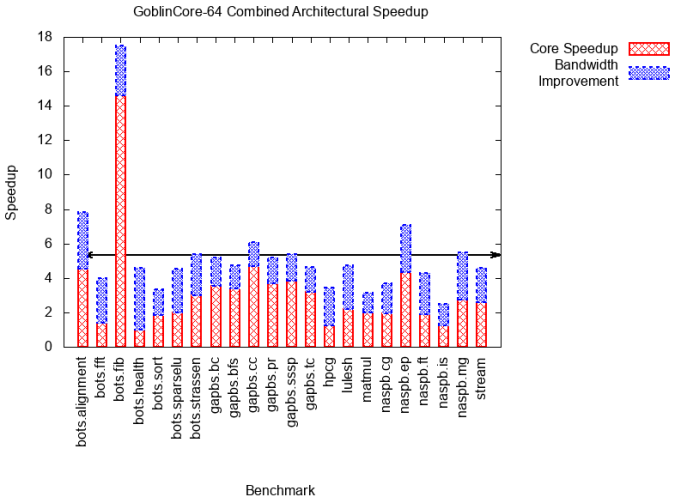

Fig. 8. Combined Architectural Speedup

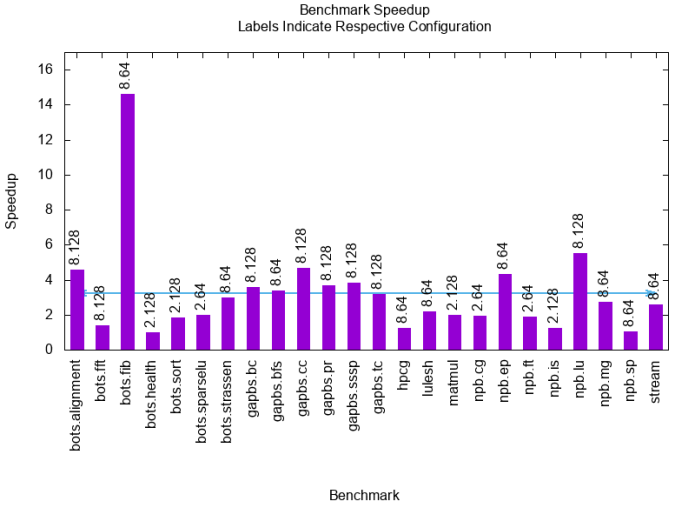

Fig. 7. Optimal Benchmark Speedup

conclude that the approach will not degrade performance across disparate workloads. Further, we find that 30% of the benchmarks achieved their peak performance with 2 tasks whereby the remaining 70% of the workloads achieved their performance with 8 tasks per core. This implies that the concurrency mechanisms present in GC64 are successful at inducing high throughput execution.

Finally, in Fig 8, we see the combined performance results when enabling context switching and memory coalescing within the GC64 infrastructure. In this figure, we plot the maximum core speedup and the average bandwidth improvement as a stacked histogram. First, we can clearly see a trend of positive speedup across all the tested workloads. This implies that our architectural techniques are successful across disparate workloads that include both dense and sparse or irregular compute and memory access patterns.

On average, we find a combined speedup of 5.33 over the baseline configurations with no GoblinCore-64 features. The application that exhibits the best potential for speedup is again the BOTS Fibonacci benchmark. This implies that as an algorithm or application begins to make significant reuse of its instruction stack or working set, we will observe a significant speedup on the GoblinCore-64 architecture. Further, for applications that exhibit very irregular memory access patterns such as those presented in the GAP Benchmark Suite, we can observe significant performance improvements over traditional monolithic core architectures.

VI. Conclusions

In conclusion, the GoblinCore-64 builds upon the base RISC-V ISA specification with hierarchical execution mechanisms, low-latency context switching mechanisms and dynamic memory coalescing in order to provide a scalable, open platform to execute high performance data intensive application payloads. Following our benchmarks and analysis, we may conclude that the GC64 architectural techniques provide a significant advantage in improving the performance, the throughput and the efficiency of data intensive and high performance computing algorithms and applications. Further, these performance advantages are measured without the use of architecture-specific programming models, compiler intrinsics or other target-specific optimizations. The benchmark codes were executed without changes to the core source code. As a result, users with disparate programming models, programming styles and algorithmic constructs may realize performance advantages using the GC64 infrastructure without changes to their source code and numerical solvers.

Acknowledgment

This work was performed within the Data Intensive Scalable Computing Laboratory (DISCL) at Texas Tech University.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

References

[1] J. D. Leidel, X. Wang, and Y. Chen, "GoblinCore-64: Architectural Specification," Texas Tech University, Tech. Rep., September 2015. [Online]. Available: http://gc64.org/wp-content/uploads/2015/09/gc64-arch-spec.pdf

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The tera computer system," *SIGARCH Comput. Archit. News*, vol. 18, no. 3b, pp. 1–6, Jun. 1990. [Online]. Available: http://doi.acm.org/10.1145/255129.255132

[3] G. Alverson, P. Briggs, S. Coatney, S. Kahan, and R. Korry, "Tera hardware-software cooperation," in *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, ser. SC '97. New York, NY, USA: ACM, 1997, pp. 1–16. [Online]. Available: http://doi.acm.org/10.1145/509593.509631

[4] A. Snavely, L. Carter, J. Boisseau, A. Majumdar, K. S. Gatlin, N. Mitchell, J. Feo, and B. Koblenz, "Multi-processor performance on the Tera MTA," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=509058.509062

[5] S. Brunett, J. Thornley, and M. Ellenbecker, "An initial evaluation of the Tera multithreaded architecture and programming system using the C3I parallel benchmark suite," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–19. [Online]. Available: http://dl.acm.org/citation.cfm?id=509058.509063

[6] C. Miyamoto and C. Lin, "Evaluating titanium spmd programs on the tera mta," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ser. SC '99. New York, NY, USA: ACM, 1999. [Online]. Available: http://doi.acm.org/10.1145/331532.331575

[7] D. Mizell and K. Maschhoff, "Early experiences with large-scale cray xmt systems," in *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, ser. IPDPS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 1–9. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2009.5161108

[8] J. D. Leidel, K. Wadleigh, J. Bolding, T. Brewer, and D. Walker, "Chomp: A framework and instruction set for latency tolerant, massively multithreaded processors," in *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, ser. SCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 232–239. [Online]. Available: http://dx.doi.org/10.1109/SC.Companion.2012.39

[9] J. D. Leidel, J. Bolding, and G. Rogers, "Toward a scalable heterogeneous runtime system for the Convey MX architecture," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 1597–1606.

[10] G. Almási, C. Caşcaval, J. G. Castaños, M. Denneau, D. Lieber, J. E. Moreira, and H. S. Warren, Jr., "Dissecting cyclops: A detailed analysis of a multithreaded architecture," *SIGARCH Comput. Archit. News*, vol. 31, no. 1, pp. 26–38, Mar. 2003. [Online]. Available: http://doi.acm.org/10.1145/773365.773369

[11] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "Toward a software infrastructure for the cyclops-64 cellular architecture," in *Proceedings of the 20th International Symposium on High-Performance Computing in an Advanced Collaborative Environment*, ser. HPCS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 9–. [Online]. Available: http://dx.doi.org/10.1109/HPCS.2006.48

[12] G. Gan, X. Wang, J. Manzano, and G. R. Gao, "Tile percolation: An openmp tile aware parallelization technique for the cyclops-64 multicore processor," in *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 839–850.

[13] J. D. Leidel, X. Wang, and Y. Chen, "Pressure-driven hardware managed thread concurrency for irregular applications," in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3'17. New York, NY, USA: ACM, 2017, pp. 7:1–7:8. [Online]. Available: http://doi.acm.org/10.1145/3149704.3149705

[14] F. Fatollahi-Fard, D. Donofrio, G. Michelogiannakis, and J. Shalf, "OpenSoC Fabric: on-chip network generator: Using chisel to generate a parameterizable on-chip interconnect fabric," in *Proceedings of the 2014 International Workshop on Network on Chip Architectures*, ser. NoCArc '14. New York, NY, USA: ACM, 2014, pp. 45–50. [Online]. Available: http://doi.acm.org/10.1145/2685342.2685351

[15] "Hybrid memory cube specification 2.0," Hybrid Memory Cube Consortium, Tech. Rep., July 2015. [Online]. Available: http://www.hybridmemorycube.org/

[16] J. D. Leidel, "GoblinCore-64: A scalable, open architecture for data intensive high performance computing," Ph.D. dissertation, Texas Tech University, 2017.

[17] X. Wang, J. D. Leidel, and Y. Chen, "Concurrent dynamic memory coalescing on goblincore-64 architecture," in *Proceedings of the Second International Symposium on Memory Systems*, ser. MEMSYS '16. New York, NY, USA: ACM, 2016, pp. 177–187. [Online]. Available: http://doi.acm.org/10.1145/2989081.2989128

[18] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-54, May 2014. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-54.html

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

[19] K. Asanovic and D. A. Patterson, "Instruction sets should be free: The case for risc-v," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2014-146, Aug 2014. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-146.html

[20] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade, "Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task parallelism in openmp," in *Proceedings of the 2009 International Conference on Parallel Processing*, ser. ICPP '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 124–131. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2009.64

[21] "Openmp application program interface version 4.0," OpenMP Architecture Review Board, Tech. Rep., July 2013. [Online]. Available: http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf

[22] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015. [Online]. Available: http://arxiv.org/abs/1508.03619

[23] D. H. Bailey, L. Dagum, E. Barszcz, and H. D. Simon, "Nas parallel benchmark results," in *Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '92. Los Alamitos, CA, USA: IEEE Computer Society Press, 1992, pp. 386–393. [Online]. Available: http://dl.acm.org/citation.cfm?id=147877.148032

[24] J. Dongarra and M. A. Heroux, "Toward a New Metric for Ranking High Performance Computing Systems," Mar. 2015.

[25] J. Dongarra, M. A. Heroux, and P. Luszczek, "Hpcg benchmark: a new metric for ranking high performance computing systems," University of Tennessee, Sandia National Laboratories, Tech. Rep., November 2015.

[26] "Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory," Tech. Rep. LLNL-TR-490254.

[27] G. Taylor, "The formation of a blast wave by a very intense explosion. ii. the atomic explosion of 1945," *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 201, no. 1065, pp. 175–186, 1950. [Online]. Available: http://rspa.royalsocietypublishing.org/content/201/1065/175

[28] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE