wrBench: Comparing Cache Architectures and Coherency Protocols on ARMv8 Many-Core Systems

Wan-Rong Gao[1], Jian-Bin Fang[1,*], Chun Huang[1], Chuan-Fu Xu[1], and Zheng Wang[2]

[1] *College of Computer Science, National University of Defense Technology, Changsha 410073, China*
[2] *School of Computing, University of Leeds, Leeds LS2 9JT, UK*

E-mail: {gaowanrong, j.fang, chunhuang, xuchuanfu}@nudt.edu.cn; z.wang5@leeds.ac.uk

Received July 15, 2018 [Month Day, Year]; accepted October 14, 2018 [Month Day, Year].

Abstract Cache performance is a critical design constraint for modern many-core systems. Since the cache often works in a "black-box" manner, it is difficult for the software to reason about the cache behavior to match the running software to the underlying hardware. To better support code optimization, we need to understand and characterize the cache behavior. While cache performance characterization is heavily studied on traditional x86 architectures, there is little work for understanding the cache implementations on emerging ARMv8 based many-cores. This paper presents a comprehensive study to evaluate cache architecture design on three representative ARMv8 multi-cores, Phytium 2000+, ThunderX2, and Kunpeng 920 (KP920). To this end, we develop the `wrBench`, a micro-benchmark suite to measure the realized latency and bandwidth of caches at different memory hierarchies when performing core-to-core communications. Our evaluation provides extensive quantified results of the cache and its coherence protocol for ARMv8 many-cores and reveals interesting undocumented features. Our paper also provides discussions and guidelines for optimizing memory access on ARMv8 many-cores.

Keywords ARMv8 Many-Cores, Cache Architecture, Microbenchmark, Core-to-Core Communication

1 Introduction

In recent years, ARMv8-based many-core CPUs are emerging as a compelling alternative for building high-performance computing (HPC) systems [1–3]. Examples of such use cases include Phytium 2000+ for China's exascale supercomputer prototype [4, 5], and `ThunderX2` in the `Astra` supercomputer [6]. Studies suggest that ARMv8-based HPC systems can achieve comparable performance as the traditional HPC hardware and are thus strong contenders in the market of next-generation HPC servers [7].

In an era where the CPU hits the memory wall [8], the cache is a key CPU component for achieving high performance. Cache performance is important for HPC many-cores because workloads running on such systems often incur frequent inter-core communications that can dominate the program execution time. To unlock the potential hardware performance, an important task of software optimization is to match the memory access pattern to the underlying cache architecture and coherence protocol. Unfortunately, doing so is non-trivial as

Regular Paper

Special Section of APPT 2021

This work is partially funded by the National Key Research and Development Program of China under Grant No. 2018YFB0204301, and the National Natural Science Foundation of China under Grant Nos. 61972408 and 61872294.

*Corresponding Author

[1]https://jcst.ict.ac.cn/EN/column/column107.shtml, May 2020.

the cache typically works as a "black box" with many implementation details unavailable to the software developers. To support code and performance optimization for many-core systems, it is highly attractive to have a way to help developers evaluate, characterize and understand the cache behavior of the underlying hardware to adapt their code accordingly.

Micro-benchmarks are an effective way of revealing the hardware implementation to allow software developers obtain hardware insights. Indeed, micro-benchmarks have been widely used to characterize and evaluate the memory hierarchy system on the conventional `x86` multi-cores. Examples of such benchmarks include the `STREAM` benchmark suite, which focuses on measuring the memory throughput, i.e., data accessing bandwidth with multi-cores [9]. The `lmbench` suite quantifies the performance of various computer components [10]. They use a pointer-chasing approach to measure the overhead of moving data across cache levels on a single core. However, `lmbench` ignores the communication overhead of transferring cachelines across different hardware cores, which is essential to optimize parallel programs concerning shared memory accesses. For this, Molka *et al.* provide a set of micro-benchmarks (`BenchIT`) to characterize such performance behaviors [11]. This tool has proven extremely valuable for quantifying core-to-core communication [12,13]. While memory performance characterization is a heavily studied field for the `x86` CPUs, there is little work for understanding the memory hierarchy design for ARMv8 high-performance many-core systems. As ARMv8 is emerging as an important class of CPUs in the HPC domain, it is desired to have a dedicated benchmark suite designed for characterizing the memory hierarchy of ARMv8 many-cores.

This work aims to close the gap of lacking ARMv8 memory characterization benchmarks. To this end, we have extended the `BenchIT` benchmark suite to adapt it to ARMv8 systems in terms of obtaining architecture parameters, setting cacheline states, enabling the clock-wise timing, and using the cache-related instructions (Section 3). Our porting leads to a new, open-source benchmark suite, namely `wrBench`[1].

We demonstrate the benefit of `wrBench` by applying it to three representative ARMv8 many-core systems: `Phytium 2000+`, `ThunderX2`, and `KP920`. We showcase that `wrBench` is useful in characterizing the underlying memory hirechy of ARMv8 systems. With wrBench, we measure the core-to-core communication performance of moving cachelines between distinct cores in terms of *latency* and *bandwidth* (Section 3). We obtain undisclosed performance data and reveal many micro-architecture details of the many-core systems on both latency (Section 4) and bandwidth (Section 5). With the extensive, quantified results in place, we compare different cache architecture design of the three ARMv8 processors. We then give quantitative guidelines for optimizing software memory accesses on ARMv8 many-core systems (Section 6).

Our evaluation results provide a quantitative reference for analyzing, modeling, and optimizing parallel programs on ARMv8 many-core systems. To the best of our knowledge, this is the first effort of systematically dissecting the memory hierarchy of ARMv8 many-core systems.

2 System Architectures

This section describes the three ARMv8 many-core CPUs target in this work. Table 1 summarizes the evluation platforms used in this work.

[1]Available at `https://github.com/WanrongGao/wrBench`

Table 1. System configuration of the three CPUs

	Phytium 2000+	**2x ThunderX2 99xx**	**2x KP920-6426**
Microarchitecture	Mars II (Phytium)	Vulcan (Cavium)	TaiShan v110 (HiSilicon)
Core frequency	2.2 GHz	2.5 GHz	2.6 GHz
Processor Interconnect	/	CCPI2	Hydra Interface
#Cores	1x 64	2x 32	2x 64
L1 cache(I/D)	32 KB/32 KB(per core)	32 KB/32 KB(per core)	64 KB/64 KB(per core)
L2 cache	2MB(per core group, shared , inclusive)	256 KB(per core)	512 KB(per core)
L3 cache	/	32 MB(per chip, shared, exclusive)	64 MB(per chip, shared)
DRAM Support	8x DDR4-2400	8x DDR4-2666	8x DDR4-2933
Operating system	Linux kernel version 4.19.46	Linux kernel version 4.19.46	Linux kernel version 4.19.46
Compiler	gcc 9.3.0	gcc 8.2.1	gcc 8.2.1

2.1 Phytium 2000+ Architecture

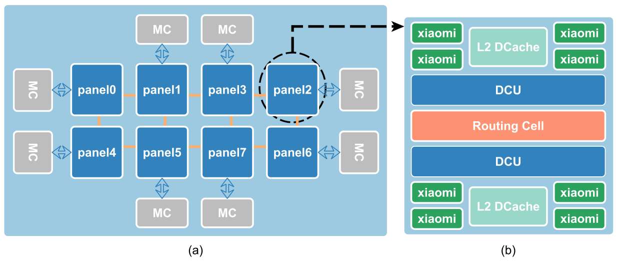


Fig.1. A high-level view of the Phytium 2000+ architecture. The 64 processor cores are groups into eight panels (a), where each panel contains eight ARMv8 based xiaomi cores (b).

Figure 1 gives a high-level view of `Phytium 2000+` based on the `Mars II` architecture. It features 64 ARMv8 compatible processing cores, which are organized into eight `panels`. Note that each panel connects a memory control unit (`MCU`).

Each panel has eight xiaomi cores, and each core has a private L1 cache of 32KB for data and instructions, respectively. Every four cores form a `core group` and share a 2MB L2 cache. Given that the L1 read port is 128 bits in width and runs at 2.2GHz, we calculate that the theoretical L1 read bandwidth is 35.2GB/s.

Each panel contains two directory control units (`DCU`) and one `routing cell`. The `DCUs` on each panel act as dictionary nodes of the entire on-chip network. With these function modules, `Mars II` conducts a hierarchical on-chip network. Phytium 2000+ uses a home-grown `Hawk` cache coherency protocol to implement a distributed directory-based global cache coherency.

2.2 ThunderX2 Architecture

ThunderX2 is built based on the `Vulcan` microarchitecture. Figure 2 shows a two-socket Vulcan system. There are 32 cores per socket operating at 2.5GHz, each with a 32KB data cache, a 32KB instruction cache, and a 256KB L2 cache. All the cores within a socket share a 32MB last level cache (L3), arranged as 2MB slices via a dual-ring on-chip bus. The L3 cache is exclusive, filling up with evicted L2 cachelines. This ring bus is connected to the 2nd-generation Cavium's Coherent Processor Interconnect (CCPI2). There are two load-store units, each capable of moving 128-bit of data per core. We calculate that the theoretical peak L1 read bandwidth is 80GB/s.

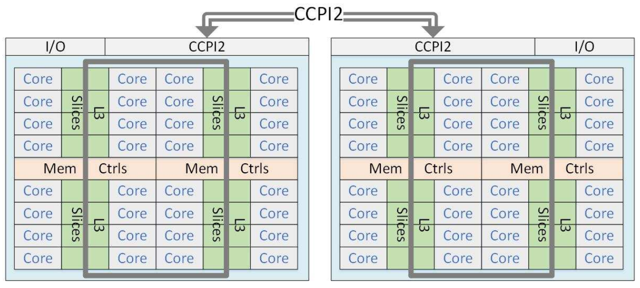


Fig.2. The ThunderX2 architecture.

2.3 KP920 Architecture

The KP920 system has two 64-bit ARMv8 processors designed by HiSilicon based on the TaiShan v110 microarchitecture. The two sockets are connected with Hydra interface ports. Each socket has two Super CPU Cluster (SCCL) and one Super IO Cluster (SICL), connected with an interchip ring bus. There are eight CPU Clusters (CCLs) within an SCCL, and each CCL has four cores running at 2.6GHz. Besides, SCCL has its memory controllers and an L3 cache slice. Each SCCL works as a NUMA node. That is, the two-socket KP920 can be seen as four NUMA nodes.

The overview of the whole TaiShan v110 microarchitecture is shown in Figure 3. Each core features 64KB private L1 instruction and data caches as well as 512KB of private L2. All the 64 cores in one SCCL share 64MB of the last level cache. Four cores within a CCL are accompanied with an L3 cache tag partition.

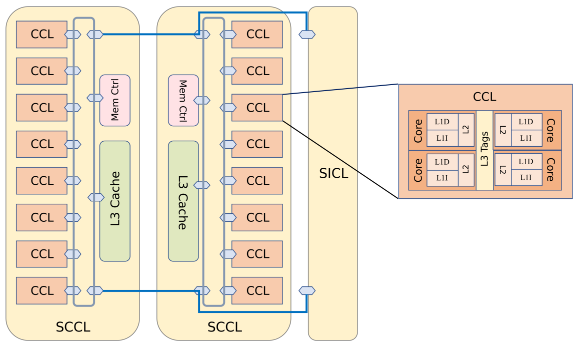


Fig.3. The KP920 architecture.

3 Benchmarking Methodology

Nowadays, many-core CPUs feature a memory system hierarchy to hide memory latencies and improve memory bandwidths. But these architectural features are transparent for programmers, and only limited software control is available. It is challenging to design micro-benchmarks that can reveal the detailed performance properties of a given cache architecture. Therefore, we carefully design a suite of memory microbenchmarks (`wrBench`) to characterize and compare the cache architectures of representative ARMv8 many-core systems.

3.1 Benchmark Design

This benchmark is extended based on the work [11, 14], mainly targeted the `x86` architectures. Due to the architecture and ISA differences between `x86` and `ARMv8`, we have heavily extended this memory benchmark to support the ARMv8 systems, aiming to be a versatile cross-architecture modeling tool for cache-coherent many-core architectures.

Overview. Figure 4 shows that `wrBench` has six measurement steps (S1–S6). We use three threads (`T0`, `Tn`, and `Tx`), each pinned to a distinct hardware core (`C0`, `Cn`, and `Cx`). `S1` ensures that all the required TLB entries for the current measurement are present in `C0`. We synchronize the threads at `S2` and `S4`. `S3` prepares data in the specified cache level of `Cn` in a well-defined coherency state (modified, exclusive, or shared). We have to flush the caches at `S5`. Because the memory benchmarks often show a mixture of effects from different cache levels rather than just one. To separate these effects, we explicitly place data in certain cache levels [14]. `S6` is the latency/bandwidth measurement step, which always runs on `C0`.

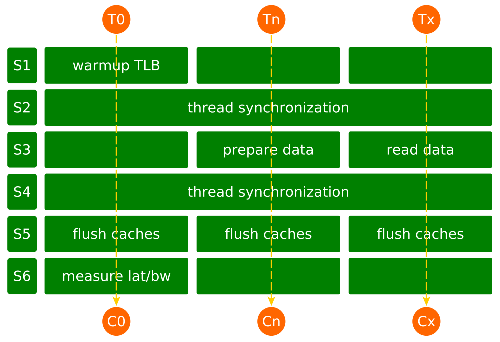


Fig. 4. The measurement steps with three threads (or cores). Note that `T0` denotes a thread running core 0 (`C0`) and `Tn` denotes a thread running on core n (`Cn`). `S1`–`S6` represent the six measurement steps, respectively.

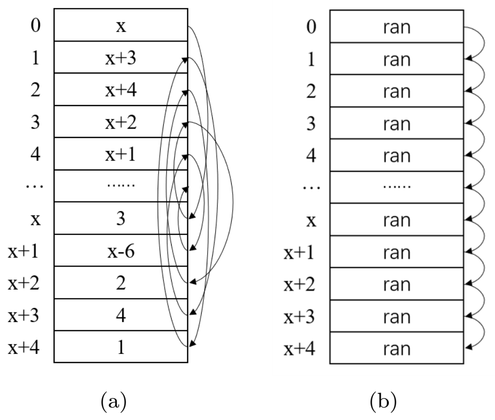


Fig. 5. Different memory access pattern supported by wrBench: accessing randomly linked data elements to measure latency (a), and accessing contiguous data elements to measure bandwidth (b). Here, "ran" represents arbitrary data contents.

We use *pointer-chasing* to measure the latency of moving a cacheline by randomly accessing discontinuous data elements (Figure 5(a)). Each cacheline is accessed only once to avoid data reuse. No consecutive cachelines are accessed to eliminate the influence of the adjacent line prefetcher. By contrast, we measure the sustainable bandwidth by continuously accessing a chunk of data elements (Figure 5(b)).

Setting cacheline states. `Tn` places data in the caches in a well-defined coherency state at `S3`. These states are generated as follows: (1) **M**odified state: `Tn` writing the data, invalidating all copies in other cores. (2) **E**xclusive state: `Tn` first writing to the memory to invalidate copies in other caches, then invalidating its cache (`dc` instruction), and then reading the data. (3) **S**hared state: `Tn` caching data in exclusive state, and then reading the data from `Cx`.

Enabling the clock-wise timing. For each measurement, we need a high-resolution timer to measure durations. We can enable the clock-wise timing with the Performance Monitors Cycle Count Register (`PMCCNTR_EL0`) on the ARMv8-based architecture. But this register is only accessible in the kernel mode. Thus, we use a kernel module to activate the performance monitoring unit. The critical steps of this kernel module are summarized as follows: (1) Reading the contents of the control register `PMCR_EL0`, (2) Activating the user mode by writing `PMUSERENR_EL0`, (3) Resetting all hardware counters by writing `PMCR_EL0`, and (4) Enabling the performance counter by writing `PMCNTENSET_EL0`. With this kernel module, the `PMCCNTR_EL0` register is accessible via the `mrs` instruction in the user mode.

Using the vector instructions. We use the vector instructions to read/write data from/to the memory system. The ARMv8-based architecture extends NEON with 32 128-bit vector registers while keeping the same mnemonics as general registers [15]. In assembly instructions, the register can identify the vector format including Vn (128-bit scalar), Vn (.2D, .4S, .8H, .16B) (128-bit vector), and Vn (.1D, .2S, .4H, .8B) (64-bit vector). We use the `ld1/st1` instruction on the ARMv8 architecture when moving data between registers and memory. The selected vector format is four single-precision floating-point words (.4S).

Using special instructions. Besides the general instructions, we use special ARMv8 instructions. `dc civac` is used to invalidate specified cachelines. It is useful when controlling the initial coherency state of cachelines. To put target data into the right cache level, we use `dmb` to ensure that the ARMv8 processors perform no optimizations on the execution order of the fetch instructions. In addition, we use the `align` instruction to avoid unaligned memory accesses.

3.2 Benchmark Portability

Our work targets the widely used ARMv8 many-core processors. This architecture is used by several high-performance computing systems, including Astra, Isambard, and Fugaku. Recently, ARM has announced the release of the ARMv9 architecture but there are currently no commercial off-the-shelf ARMv9 processors available. We believe wrBench can be easily ported to ARMv9. Doing so would require using ARMv9-specific assembly instructions for loads and stores as well as providing routines for obtaining system parameters on frequency and cache organization. Other than these, the majority part of wrBench can remain unchanged. Our future work will look into the memory characterization of ARMv9.

4 Latency Results

In this section, we analyze and compare the latency results of the three ARMv8 architectures. We measure the latency of `c0` loading cachelines which are `exclusive`, `modified`, or `shared` in different cores and different cache levels. The data set size is set from half of the L1 cache (16KB or 32KB) to 200MB to cover each memory level. We find that the latency results show a visible phase change as the size of the data set increases. And this phase change is consistent with the capacity of each cache level.

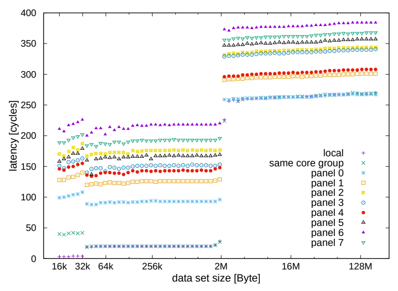
4.1 On Phytium 2000+

The cores on Phytium 2000+ are organized into eight panels. We measure the latency when `c0` is accessing cores on panel1 to panel7, respectively. For each panel, we choose to use the first core. Besides, the "local" latency means accessing data that has been prepared in `c0` locally. The "same core group" means the accessed core and caches are located on the same core group with `c0`, sharing the same L2 cache slice. The results are shown in Figure 6 and Table 2.

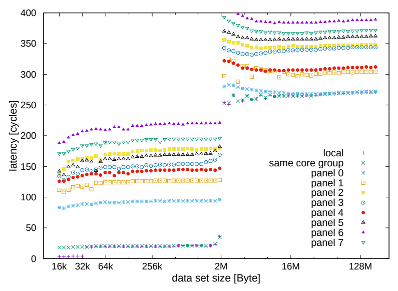
Local accesses. From Figure 6, we see that the local accessing latency is independent of the coherency state of the accessed data. The local latency changes twice during the whole process, i.e., at 32KB (the size of L1 cache) and 2MB (the size of L2 cache). The latencies of accessing the local L1 and L2 cache are 1.8ns (4cycles) and 9.1ns (20cycles), respectively. The specification of `Mars I` describes that accessing the local L1 and L2 takes 2ns and 8ns, respectively, which is identical to our measured results [3].

Within a core group. Every four cores on Phytium 2000+ share a local L2 cache slice and form a core group. Thus, the accessing latency to the L2 cache is the same as the local one (9.1ns). For the remote L1 cache, we observe the latency reduces from 18.6ns to 9.1ns when the cacheline is *shared*. This change shows that `c0` can directly obtain data from the L2 cache. It can be inferred that the L2 cache is inclusive. Cachlines can be modified in the L1 cache without being written back to the L2 cache because of the write-back policy adopted on Phytium 2000+. This feature leads to a larger overhead (18.6ns versus 9.1ns) when accessing the *modified* data located in the remote L1.

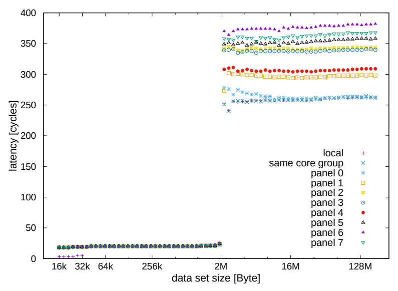
Across core group. The hardware cores on a different core group from `c0` will not share the same L2 cache slice. Accessing data across these cores must be forwarded by the `routing cell`. As a result, the latency numbers will be larger. The specific latency numbers are determined by the distance of these core groups to



(a) Exclusive or Modified



(b) Shared (other cores)



(c) Shared (`c3`)

Fig.6. Read latencies for accessing different locations on Phytium 2000+.

Table 2. Latencies (ns(cycle)) for accessing different locations on Phytium 2000+

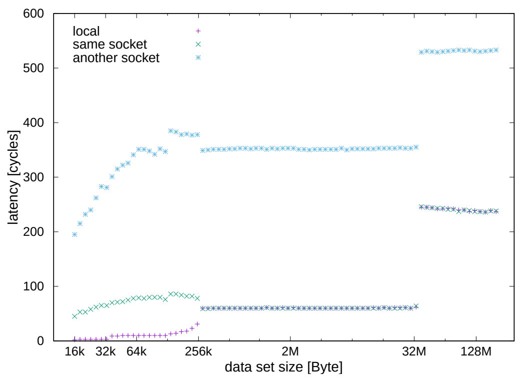
	Exclusive/Modified		Shared(other cores)		RAM
	L1	L2	L1	L2	
local	1.8(4)	9.1(20)	1.8(4)	9.1(20)	122.3(269)
same core group(c1)	18.6(41)		9.1(20)		
panel 0(c4)	45(99)-49.1(108)	42.3(93)	37.3(82)-39.5(87)	42.3(93)	
panel 1(c8)	53.6(118)-59.5(131)	54.1(119)	44.5(98)-50.9(112)	54.1(119)	138.2(304)
panel 2(c16)	75.5(166)-80.5(177)	76.3(168)	68.2(150)-72.3(159)	76.3(168)	158.2(348)
panel 3(c24)	65.5(144)-70.5(155)	65.5(144)	57.7(127)-61.8(136)	65.5(144)	154.6(340)
panel 4(c32)	62.7(138)-67.3(148)	61.4(135)	53.6(118)-58.2(128)	61.4(135)	140(308)
panel 5(c40)	70.9(156)-77.3(170)	72.7(160)	60.5(133)-88.8(147)	72.7(160)	162.7(358)
panel 6(c48)	92.3(203)-99.1(218)	95.5(210)	80.9(178)-87.3(192)	95.5(210)	174.5(384)
panel 7(c56)	82.7(182)-88.6(195)	84.5(186)	74.5(164)-80(176)	84.5(186)	167.3(368)

`c0`. The latency difference between `c0` accessing the two core groups on a remote panel is around 3ns. We choose to use the core group with a smaller latency to represent the entire panel in this context. It is worth noting that Phytium 2000+ adopts a unique strategy when accessing *shared* cachelines. `c0` obtains data neither from the most recently visited copy (like the MESIF protocol) nor from the nearest copy (the strategy used by ThunderX2). If a third copy is in the same core group with `c0`, it can be obtained directly from the shared local L2 cache . In this situation, when the size of the data set is smaller than the L2 cache, the latencies to access data are equal to the local L2 latency (9.1ns). The data beyond the L2 cache size can only be obtained from the remote memory module. Thus, the latency shows a leap at 2MB, displayed in Figure 6(c). Otherwise, data can be obtained only from the first copy rather than a closer copy (Figure 6(b)). The latencies of accessing the *shared* cachelines in the remote L2 caches are consistent with the *exclusive*. Besides, the cores on the same panel are connected directly to the same memory module and incur a similar latency. The latencies to other panels increase over the panel distance.

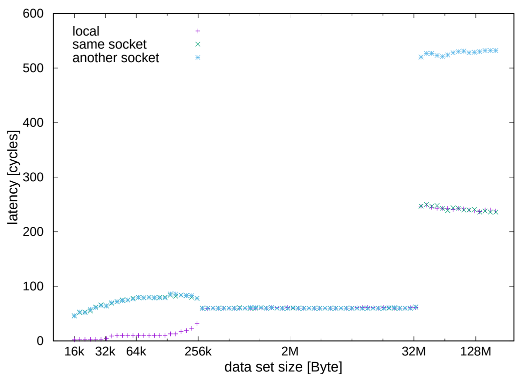
4.2 On ThunderX2

The latency measurement results on ThunderX2 are shown in Figure 7 and Table 3. The "local" has the same meaning as that in Section 4.1. The "same socket" refers to loading data from cores that share the same L3 cache with `c0`. And here, we choose to use `c1`. The results labeled as "another socket" denote accesses to data that is located in the core-caches of the other socket. And here we choose to use `c32`.

Local accesses. The three turning points of the local latency are consistent with the sizes of the three cache levels of ThunderX2. The latencies are 1.2ns (3cycles),



(a) Exclusive or Modified or Shared(`c40`)



(b) Shared(`c8`)

Fig.7. Read latencies for accessing different locations on ThunderX2.

Table 3. Latencies (ns(cycle)) for accessing different locations on ThunderX2

	Exclusive/Modified/Shared(c40)			Shared(c8)			RAM
	L1	L2	L3	L1	L2	L3	
local	1.2(3)	4.00(10)	24.00(60)	1.2(3)	4.00(10)	24.00(60)	95.6(239)
same socket	18.0(45)-26.0(65)	∼31.2(78)		18.0(45)-26.0(65)	∼31.2(78)		
another socket	78.0(195)-112.4(281)	∼140.7(352)	140.7(352)				212.3(531)

4ns (10cycles), and 24ns (60cycles), respectively. These results are consistent with the numbers we measured with `lmbench` (1.6ns, 4.4ns and 25.8ns).

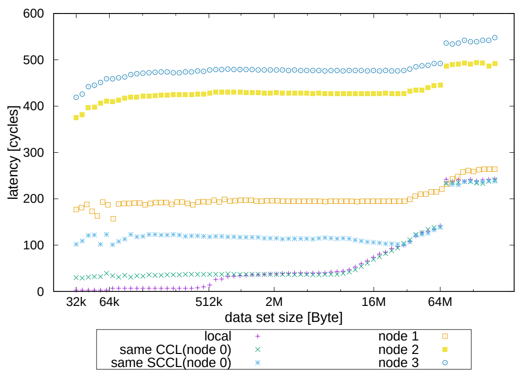
Within a socket. As `c1` shares the same L3 slice with `c0`, the data located in the L3 cache of `c1` can be accessed directly while accessing the local L3 cache (24ns). Since L3 in ThunderX2 is exclusive, it does not contain data placed in the higher caches. Therefore, when `c0` accesses data in the remote L1 or L2 caches, it must first load data from the higher-level caches. This operation is independent of the coherency states (7.2ns).

Another socket. Access to another socket is through the CCPI2 link. Transferring data from the L3 cache of `c32` takes around 140.7ns (352cycles). We obtain that the latency of walking through this link is 116.7ns by comparing the latency numbers of accessing `c1` and `c32`. When the cachelines are shared with `c8` (Figure 7(b)), the latencies of loading them from caches of `c32` become the same as that from `c1`. When the second copy is placed on `c40` (Figure 7(a)), transferring the *shared* cachelines has no difference from the *exclusive* state. These indicate that the memory controller is able to fetch the nearest copy.

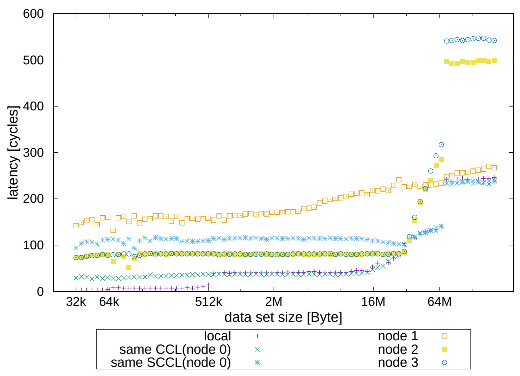
4.3 On KP920

As we have shown in Section 2.3, KP920 has four NUMA nodes. To measure the latency across NUMA nodes, we choose to use the first core of each remote node. We also measure the latency numbers of `c0` accessing `c0` (local), `c1` (the same CCL), and `c4` (the same SCCL) within a NUMA node. The results are shown in Figure 8 and Table 4. It should be noted that the L3 columns in the table only lists stable values.

Within a NUMA node. The first two turning points of the local latency occur at 64KB and 512KB, i.e., the private L1 and L2 cache size per core. The last change is at 64MB, which is the LLC size on a socket. The accessing latencies of the local L1 and L2 caches are 1.15ns (3cycles) and 2.7ns (7cycles), respectively.



(a) Exclusive or Modified



(b) Shared(node 0)

Fig.8. Read latencies for accessing different locations on KP920.

Table 4. Latencies (ns(cycle)) for accessing different locations on KP920

	Exclusive or Modified			Shared			RAM
	L1	L2	L3	L1	L2	L3	
local	1.15(3)	2.7(7)	14.2(37)	1.15(3)	2.7(7)	14.2(37)	91.5(238)
same CCL	11.9(31)	14.2(37)		11.9(31)	14.2(37)		
same SCCL	39.2(102)-45(122)	45(122)	44.2(115)	39.2(102)-45(122)	45(122)	44.2(115)	
node 1	68.1(177)-75(195)	75(195)		43.8(144)-61.5(160)	61.5(160)	61.5(160) -75(195)	102.3(264)
node 2	146.9(382)-158.1(411)	164.2(427)		28.1(73)-30.4(79)	31.2(81)		189.2(492)
node 3	161.2(419)-176.5(459)	183.5(477)					208.5(542)

The `lmbench` measurement results are 1.5ns and 3.1ns, which are basically consistent with ours. For the remote L1 and L2 caches, we observe that the latencies are close to accessing the corresponding L3 caches. This observation indicates that the L3 cache of KP920 is inclusive. As shown in Figure 8, the latency of accessing L3 varies a lot. The specific changing process is shown in Table 5. `c1` is suited in the same CCL with `c0`, sharing the same L3 Tag Partition. Thus, its latency should be the same as "local" in the L3 stage. We mainly compare "local" and "same SCCL" latency (Figure 8(a)). We see that both of them keep steady before 8MB. After that, the former increases (from 14.2ns to 38.5ns) as the dataset grows while the latter decreases (from 44.2ns to 38.5ns). Finally, they meet at 32MB and continue to increase till 64MB. The difference before 32MB illustrates the LLC of KP920 has an affinity to the cores. That is, different CCLs correspond to different L3 cache slices. The accessed data is prioritized to be placed in their corresponding local L3 slice, which differs from ThunderX2.

Table 5. Latencies (ns(cycle)) for accessing LLC on KP920

	-8MB	8-32MB	32-64MB
local	14.2(37)	14.2(37)-38.5(100)	38.5(100)-53.5(139)
same CCL			
same SCCL	44.2(115)	44.2(115)-38.5(100)	
node 1	75(195)		75(195)-85(221)
node 2	164.2(427)		164.2(427)-171.2(445)
node 3	183.5(477)		189.2(492)-183.5(477)

The latency of accessing the L3 cache on ThunderX2 does not vary from core to core, while the LLC is also divided into slices. The latencies eventually reach the same before the whole 32MB LLC is filled. Thereafter, the data will be placed in another L3 cache on node1. The more dataset is over 32MB, the larger overhead the remote access incurs. The latency increases from 38.5ns to 53.5ns.

Across NUMA nodes. Accessing *exclusive* or *modified* cachelines on the remote nodes requires walking through the interchip bus (node1) or the Hydra link (node2 and node3). As a result, the latency numbers are larger compared with "same SCCL". From Table 4, we see that the latencies across SCCLs and sockets are approximately 10.8ns and 86.9ns, respectively.

When the cachelines are *shared* (suppose that the second copy is in node0), the latencies across the NUMA node show a significant difference (Figure 8(b)). The latencies of accessing cores in another socket (node2 and node3) decrease significantly to the same value (31.2ns). It is even smaller than the latency of accessing "same SCCL" (local node, 44.2ns). It is possible because `c0` accesses the copy in node0 rather than nodes of another socket. For a core on the same socket but in another SCCL (node1), the latency also decreases but with a very small difference. It does not reach the value of accessing the local node. We argue that the data is still transferred through the interchip bus.

5 Bandwidth Results

In this section, we focus on the read bandwidth. The experimental settings are the same as those for the latency measurement. Our following analysis will show that the read bandwidth results are consistent with the latency results, with only several exceptions.

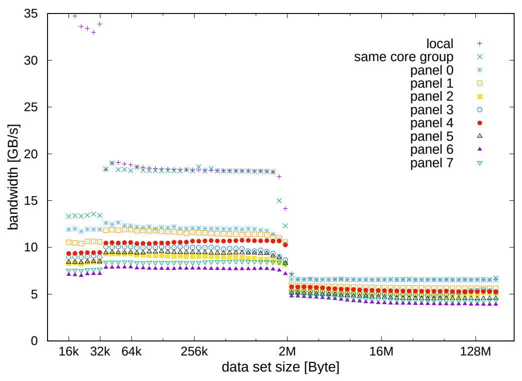
5.1 On Phytium 2000+

As shown in Figure 9, the read bandwidth of the local L1 cache is 33.6 GB/s, which is close to its theoretical peak (35.2 GB/s). Meanwhile, reading data from the local L2 cache can reach a bandwidth of 18.2 GB/s. The read bandwidth to the L2 cache of `c1` is the same as that reading from the local L2 cache of `c0`. It is because the two cores share the same L2 cache slices. But the bandwidth is reduced to be around 13.3 GB/s when `c0` loads *exclusive* or *modified* cachelines suited in `c1`'s L1 cache. This is because a check operation is required. The bandwidths of accessing the L1 and L2 caches of the remote cores have a similar trend. The specific bandwidth numbers can be found in Table 6. If the cachelines are *shared*, it is unnecessary to perform this check step. Thus, the remote L1 bandwidth reaches the same number as that accessing the corresponding L2 cache.

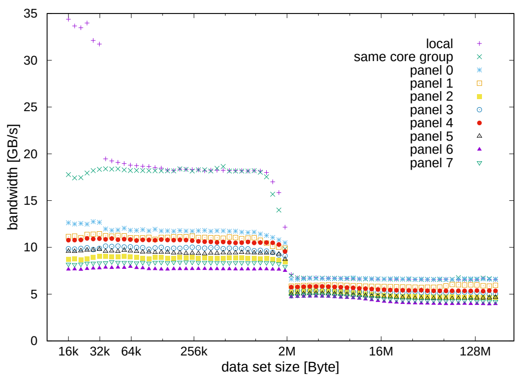
Table 6. Bandwidth (GB/s) on Phytium 2000+

	Exclusive/Modified		Shared		RAM
	L1	L2	L1	L2	
local	33.6	18.2	33.6	18.2	6.5
same core group	13.3		13.3		
panel 0	11.9	12.5	12.5	11.8	
panel 1	10.5	11.5	11.4	11	5.6
panel 2	8.3	9	8.8	9	5
panel 3	9	10	9.8	10	5
panel 4	9.4	10.5	10.8	10.5	5.3
panel 5	8.5	9.5	9.6	9.5	4.5
panel 6	6.9	7.8	7.7	7.8	4
panel 7	7.6	8.4	8.3	8.4	4.3

Because `c0`, `c1`, `c4` are located on the same panel, they are connected directly to the same memory module. When accessing data in the local memory module, the bandwidth can reach around 6.5 GB/s. The bandwidth of accessing the remote memory modules varies from panel to panel. The farther a panel is located from panel 0, the smaller bandwidth we will have.

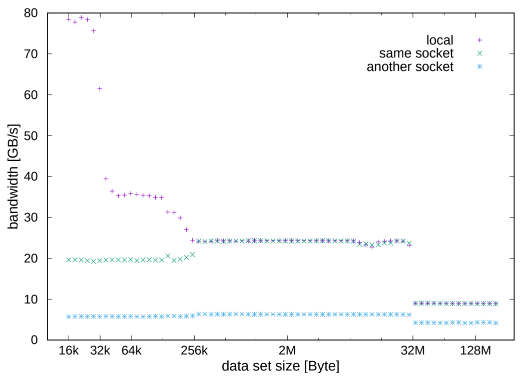


(a) Exclusive or Modified

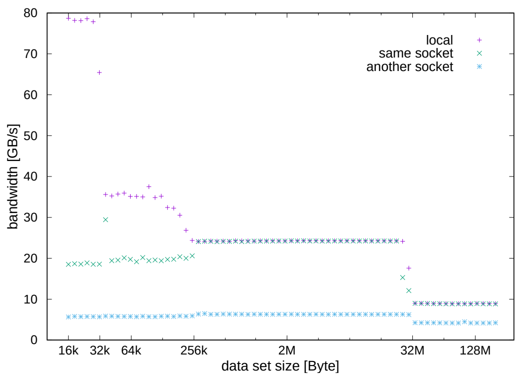


(b) Shared

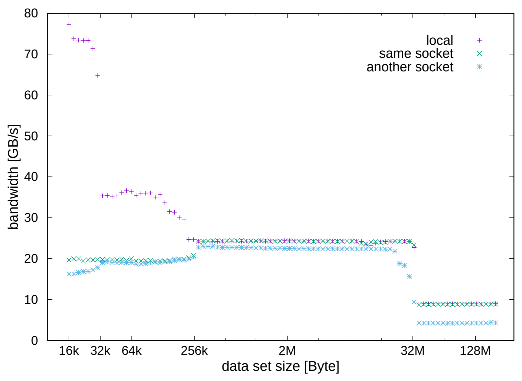
Fig.9. Read bandwidth for accessing different locations on Phytium 2000+.



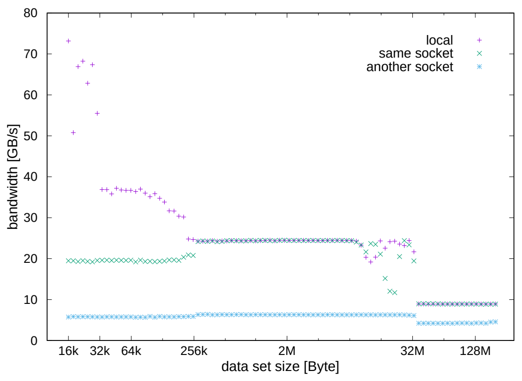
(a) Exclusive



(b) Modified



(c) Shared(`c8`)



(d) Shared(`c40`)

Fig.10. Read bandwidth for accessing different locations on ThunderX2.

Table 7. Bandwidth (GB/s) on ThunderX2

	Exclusive			Modified			Shared(c8)			Shared(c40)			RAM
	L1	L2	L3	L1	L2	L3	L1	L2	L3	L1	L2	L3	
c0	78.3	35.5	24.3	78.3	35.5	24.3	73.3	35.5	24.3	67.8	51.7	24.3	9
c1	19.5			18.5	19.5		19.5			19.5			
c32	5.8		6.3	5.8		6.3	16.2-17.7	19	22.4	5.8		6.3	4.2

5.2 On ThunderX2

Figure 10 and Table 7 give a high-level view of the bandwidth numbers on ThunderX2.

The read bandwidth to the local L1 cache can reach

78.3 GB/s. The measurement result is basically consistent with the theoretical value (80 GB/s). Reading data from the local L2 cache and L3 cache can reach a bandwidth of 35.5 GB/s and 24.3 GB/s, respectively. However, the local L1 read bandwidth drops when accessing the *shared* cachelines (73.3 GB/s on `c8` and 67.8 GB/s on `c40`). And it fluctuates significantly while another copy is suited on the remote socket.

As we have mentioned above, only when the accessed data is located in the L3 cache, `c0` can load data from the L3 cache directly. In such a case, the read bandwidth to `c1` is the same as accessing the local L3 cache slice, reaching 24.3 GB/s. Otherwise, the data must be loaded from the remote L1 or L2 caches. When the cacheline is *exclusive* or *shared*, it can be loaded directly from the remote L2 caches (19.5 GB/s). When the cacheline is *modified*, it has to be loaded from the remote higher level cache (18.5 GB/s).

`c32` is located on another socket, not sharing a common L3 cache slice with `c0`. As a result, the read bandwidth of accessing L1 or L2 cache of `c32` is 5.8 GB/s, and accessing L3 yields is larger bandwidth, staying around 6.3 GB/s. The bandwidth of accessing data in the local memory module can reach 9 GB/s, whereas accessing data from another memory module stays around 4.2 GB/s.

5.3 On KP920

The specific bandwidth numbers are listed in Table 8. The local L1 bandwidth is 81.2GB/s, and the L2 bandwidth is 51.8GB/s. It can be seen from Figure 11 that the L3 bandwidth in node0 exhibits a complicated trend. For “local” and “same CCL” cores (`c0` and `c1`), the bandwidth first stays at 21.4GB/s and then decreases to 16.5GB/s at 32MB. On the contrary, the bandwidth of “same CCL” (`c4`) first stabilizes at 14.7GB/s and then increases to 16.5GB/s. From 32MB to 64MB, both of them decrease to the memory bandwidth (11.6GB/s). The variation of bandwidth is due to the affinity of the L3 cache slices, as we have analyzed in Section 4.3.

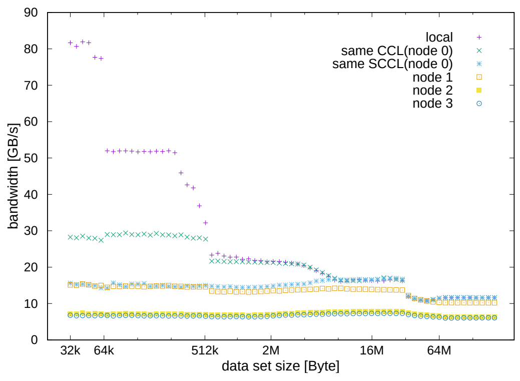
The “node1” core (`c32`) is located on another SCCL with `c0` but still within the same socket. Therefore, the bandwidth loading data from the remote caches is almost the same as `c4`. Similarly, the bandwidth of “node2” is close to “node3”, while they are also in the same socket. These explain that the interchip connections within a socket do not affect the bandwidth performance.

When cachelines are *shared* (Figure 11(b)) , `c0` can load data from a copy in a local node rather than one from a different socket. So the bandwidth of the remote caches in node2 or node3 can reach the same as “same CCL”. While the accessed node is in the same socket with `c0` (node1), the bandwidth result shows that `c0` still uses the copy in the remote node rather than the local node.

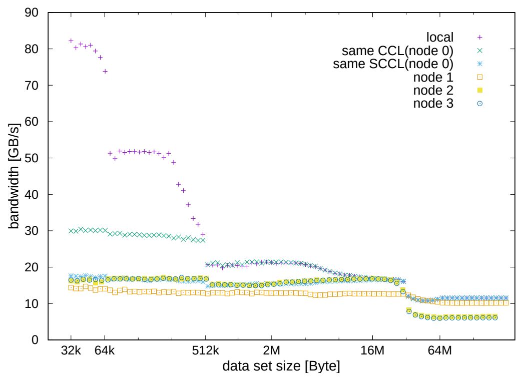
The bandwidth of accessing the local memory module is 11.6 GB/s. When accessing the remote memory module on other nodes, the bandwidth will decrease significantly. The difference in memory bandwidth within a socket is much smaller than the across-socket one. The former is as low as 1.3 GB/s (11.6 GB/s vs 10.3 GB/s), while the latter can reach 5.2 GB/s (11.6 GB/s vs 6.4 GB/s).

6 Discussion

Our results have revealed significant differences in intra-core and inter-core communication performance of the three ARMv8 many-core systems. Their performance results are compared in terms of the cache organization and the coherency protocols in Section 6.1. We then summarize optimization guidelines based on the comparison and analysis (Section 6.2).



(a) Exclusive or Modified



(b) Shared(node 0)

Fig.11. Read bandwidth for accessing different locations on KP920.

Table 8. Bandwidth (GB/s) on KP920

	Exclusive or Modified			Shared			RAM
	L1	L2	L3	L1	L2	L3	
local	81.2	51.8	21.4-16.5	81.2	51.8	21.4-16.5	11.6
same CCL	28						
same SCCL	15.3	14.7	14.7-16.5	17.5	16.8	15.2-16.5	
node 1			13.3-14.0	14.1	13.3	13.2-12.8	10.3
node 2	7		7-7.8	16.4	16.8	15.2-16.5	6.4
node 3	6.7		6.4-7.3				6.1

6.1 Comparing Communication Performance

Intra-Core Cache Organization. Each core of Phytium 2000+ or ThunderX2 has a private 32KB L1 data cache. KP920 owns a larger private L1 data cache per core, twice as large as the former. Accessing the local L1 cache on the three platforms is very fast, taking around 3 cycles. But in terms of the read bandwidth, Phytium 2000+ can achieve around half of that on the other two platforms (33.6GB/s on Phytium 2000+ vs. 78.3GB/s on ThunderX2 and 81.2GB/s on KP920). It is because Phytium 2000+ can load 32 bytes per cycle, while ThunderX2 and KP920 can load 64 bytes per cycle. ThunderX2 and KP920 have private L2 cache slices. By contrast, the L2 cache of Phytium 2000+ is a shared last-level cache, which will be discussed in the following subsection. Both the local L2 latencies of ThunderX2 and KP920 are small, 4ns and 2.7ns, respectively. But the L2 size of ThunderX2 is only half of that of KP920. Moreover, the L2 read bandwidth of KP920 is much larger than that on Thunder (51.8 GB/s vs. 35.5GB/s). Thus, in general, the performance of KP920's L2 cache is better than that of ThunderX2.

Inter-Core Cache Organization. We compare the shared LLC cache organization and analyze the inter-core LLC latencies. Table 9 summarizes the LLC latencies for `c0` accessing other core on three ARMv8 platforms. In addition, we measure the inter-core LLC latency between 64 cores. We use heat maps to visualize the measurement results in Figure 12, where the three platforms use a uniform color band for the intuitive comparison. It is easy to see that the LLC latency between cores is symmetrical. The LLC size of Phytium 2000+ is the smallest (i.e., 2MB sharing among four cores and thus 32MB in total), while the latency is minimal (9.1ns). The latency of the local LLC on Phytium 2000+ is minimal (9.1ns) because there is only one private cache between the core and the local

Table 9. LLC latencies (ns) for `c0` accessing other cores on three platforms

core	0-3	4-7	8-15	16-23	24-31	32-39	40-47	48-55	56-63
Phytium 2000+	9.1	42.3	54.1	76.3	65.5	61.4	72.7	95.5	84.5
ThunderX2	24					140.7			
KP920	14.2-38.5-53.5	44.2-38.5-53.5				75-85			

LLC. But its size is too small, with 2MB sharing by four cores. When a core accesses other LLCs, the latency varies from 42.3ns to 95.5ns according to the distance between panels. On ThunderX2, each core can access another core sharing with the LLC with the same latency (24ns). It is because the 32 cores on a single chip are connected through a uniform ring bus. The latency number increases to 140.7ns when accessing the LLC on another socket, which is the largest among the three platforms. Contrary to ThunderX2, the LLC of KP920 is partitioned and has an affinity to hardware cores, resulting in nonuniform latency. In general, the latency can be divided into three levels according to the core layout of CCL, SCCL, and socket. Its advantage compared with ThunderX2 is that the 64 cores are located on the same socket, so the maximum latency is smaller. But due to the affinity, its LLC latency is unstable as the data size grows, which has been analyzed in Section 4.3.

Cache-Coherency Protocols. We compare the three systems in terms of coherency protocols. We observe that the three processors show no difference between the *exclusive* and the *modified* states. It is probably because they all use directory-based protocols. The most noticeable difference between them is how they handle the *shared* data. When there are multiple *shared* copies on distinct cores, ThunderX2 adopts a straightforward policy – the accessing core can obtain data from the nearest copy. Meanwhile, Phytium 2000+ will fetch data from the first copy if no copy is in the same core group with the accessing core. It may lead to a large latency because the first copy can have the farthest distance. Besides, ThunderX2 uses an exclusive LLC policy when managing the multi-level caches. In this case, cachelines from the remote higher-level caches must be fetched from the remote cores or main memory. It is because no copy exists in the exclusive L3 cache. From the measurement results (Figure 7(a)), we observe that it chooses to use the former. Compared with the inclusive policy adopted by KP920, ThunderX2 shows no performance loss but increases the effective capacity of the relevant cache slices.

6.2 Optimization Suggestions

Based on our measurement results and the analysis, we summarize three optimization suggestions for programmers on the three ARMv8 many-core systems.

OS1: Our performance numbers can be used to identify the communication bottleneck of the parallel algorithms. The efficiency of inter-core communication on many-core processors is an important factor restricting the performance of parallel programs. Identifying the communication bottleneck is the basis of optimizing parallel algorithms. Through abstracting inter-core communication into the read and write operations on shared variables, we can build a communication model for the ARMv8 platforms with the latency numbers measured from wrBench. For example, the synchronization barrier is a typical parallel algorithm restricted by inter-core communication. Different barrier algorithms such as dissemination and tournament algorithms have different communication patterns. We can use the communication model to determine the bottleneck of each algorithm on the ARMv8 platforms for the

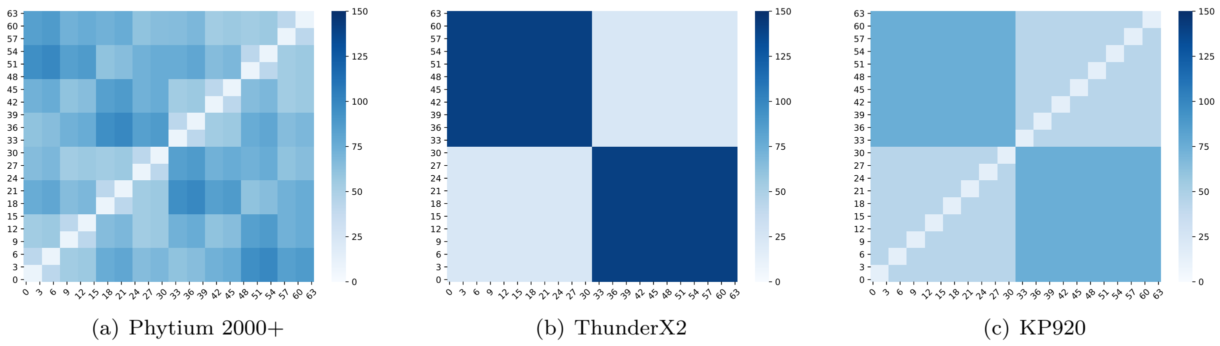

(a) Phytium 2000+ (b) ThunderX2 (c) KP920

Fig.12. Core-to-core LLC latencies (ns) for 64 cores on three platforms. Lighter color represents a smaller latency.

following optimization.

OS2: We recommend that programmers pin a group of threads that access the same shared variables to cores that share the same LLC slices. Our experimental results show that the cost of obtaining data from cores sharing the same LLC slices is generally smaller than other cores. Taking the tree-based barrier as an example, a group of threads shares a shared integer variable to achieve synchronization. Controlling each group of threads shares the same LLC slices as far as possible can avoid expensive remote accesses. The same idea can also be used to optimize other collective communication algorithms such as broadcast and reduction. In particular, some processors like KP920 may have non-uniform LLCs. Therefore, it is better to place threads in four cores in a CCL. And the size of the data shared by multiple threads should be controlled within 8MB for fast local accesses.

OS3: Programmers should keep an eye on the panel distances. When there must be access across NUMA nodes, it is crucial to select the right nodes to minimize the cross-node overhead. Taking running SpMV (sparse matrix-vector multiplications) on Phytium 2000+ as an example, programmers should pay attention to the distance between panels. It is because using multiple threads for SpMV involves core-to-core communication to achieve the sharing of the dense input vector. Many hypergraph-based algorithms have been proposed to minimize the inter-thread communication.

7 Related Work

Although the effective use of the memory systems is essential to obtain the best performance, vendors seldom provide the details of the memory hierarchy or the achieved performance. For this reason, researchers have to obtain such performance results and implementation details through measurements.

Babka *et al.* [16] propose experiments that investigate detailed parameters of the `x86` processors. The experiment is built on a general benchmark framework and obtains the required memory parameters by performing one or a combination of multiple open-source benchmarks. It focuses on detailed parameters, including the address translation miss penalties, the parameters of the additional translation caches, the size of cacheline, and the cache miss penalties.

McCalpin *et al.* [9] present four benchmark kernels (Copy, Scale, Add, and Triad), `STREAM`, to access memory bandwidth for current computers, including uniprocessors, vector processors, shared-memory systems, and distributed-memory systems. `STREAM` is one of the most commonly used memory bandwidth measurement tools in Fortran and C. But it focuses on throughput measurement without considering the latency metric.

Molka *et al.* [11] propose a set of benchmarks, including studying the performance details of the `Nehalem` architecture. Based on these benchmarks, they obtain undocumented performance data and architectural properties. It is the first to to measure the core-to-core communication overhead, but it is only applicable to the `x86` architectures. Fang *et al.* extend the microkernels to Intel Xeon Phi [13]. Ramos *et al.* [12] propose a state-based modeling approach for memory communication, allowing algorithm designers to abstract away from the architecture and the detailed cache coherency protocols. The model is built based on the measurement numbers of the cache-coherent memory hierarchy.

Besides the x86 processors, researchers have designed microbenchmarks for other many-core processors to demystify their microarchitectures and memory hierarchies. Wong *et al.* [17] developed a set of CUDA microbenchmarks and measured the architectural characteristics of the NVIDIA GT200 (GTX280) GPU. Mei *et al.* [18] proposed a fine-grained pointer chasing microbenchmark to investigate the throughput and access latency of GPU's global memory and shared memory. They investigated the GPU memory hierarchy of three recent NVIDIA GPUs: Fermi, Kepler, and Maxwell. Lin *et al.* [19] presented a microbenchmark suite called `swCandle` to evaluate the key micro-architectural features of the SW26010 many-core processor. This evaluation work targets specialized accelerators, e.g., GPGPUs or SW26010, rather than the cache-coherent many-core architectures.

There exist some performance analysis works on the ARMv8-based high-performance computing (HPC) systems. Mantovani *et al.* [20] analyzed the performance and energy consumption of Dibona, a system powered by ThunderX2. Simon *et al.* [21] presented performance results of Isambard, which combines ThunderX2 CPUs with Cray's Aries interconnect. These works focus on the performance behaviors of the entire system rather than the cache architectures.

8 Conclusion

The ARMv8 many-core architectures have been widely used to build the current and next-generation HPC systems, but they feature a variety of cache organizations and coherency protocols, which makes them complicated to understand and hard-to-use. Given that the cache architecture is a critical factor affecting overall performance, it is significant to understand its working mechanism behind the "black-box". This article focuses on comparing the cache architecture and its coherent protocols of three representative ARMv8 architectures with microbenchmarks. For this, we have heavily extended and refined a set of benchmarks (`wrBench`) to measure the intra-core and inter-core communication performance of the ARMv8 systems.

We have chosen three representative ARMv8 systems as our experimental platforms to demonstrate the potential of `wrBench`, including Phytium 2000+, ThunderX2, and KP920. Experimental results show that our `wrBench` can provide a detailed and quantitative performance description of the ARMv8 many-core memory hierarchy. By comparing and analyzing the communication performance, we summarize and analyze the pros and cons of the three ARMv8 processors in cache organization and coherency protocol. We also provide guidelines on improving the performance of parallel programs by optimizing memory accesses.

For future work, we will extend our wrBench to ARMv9 machines once they are available.

References

[1] M. A. Laurenzano, A. Tiwari, A. Cauble-Chantrenne, A. Jundt, W. A. W. Jr., R. L. Campbell, and L. Carrington,

"Characterization and bottleneck analysis of a 64-bit armv8 platform," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2016, Uppsala, Sweden, April 17-19, 2016*. IEEE Computer Society, 2016, pp. 36–45.

[2] N. Stephens, "Armv8-a next-generation vector architecture for HPC," in *2016 IEEE Hot Chips 28 Symposium (HCS), Cupertino, CA, USA, August 21-23, 2016*. IEEE, 2016, pp. 1–31.

[3] C. Zhang, "Mars: A 64-core armv8 processor," in *2015 IEEE Hot Chips 27 Symposium (HCS)*. IEEE, 2015, pp. 1–23.

[4] X. You, H. Yang, Z. Luan, Y. Liu, and D. Qian, "Performance evaluation and analysis of linear algebra kernels in the prototype tianhe-3 cluster," in *Supercomputing Frontiers - 5th Asian Conference, SCFA 2019, Singapore, March 11-14, 2019, Proceedings*, ser. Lecture Notes in Computer Science, D. Abramson and B. R. de Supinski, Eds., vol. 11416. Springer, 2019, pp. 86–105.

[5] J. Fang, X. Liao, C. Huang, and D. Dong, "Performance evaluation of memory-centric armv8 many-core architectures: A case study with phytium 2000+," *J. Comput. Sci. Technol.*, vol. 36, no. 1, pp. 33–43, 2021.

[6] K. T. Pedretti, A. J. Younge, S. D. Hammond, J. H. L. III, M. L. Curry, M. J. Aguilar, R. J. Hoekstra, and R. Brightwell, "Chronicles of astra: challenges and lessons from the first petascale arm supercomputer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, C. Cuicchi, I. Qualters, and W. T. Kramer, Eds. IEEE/ACM, 2020, p. 48.

[7] F. Mantovani, M. Garcia-Gasulla, J. Gracia, E. Stafford, F. Banchelli, M. Josep-Fabrego, J. Criado-Ledesma, and M. Nachtmann, "Performance and energy consumption of HPC workloads on a cluster based on arm thunderx2 CPU," *Future Gener. Comput. Syst.*, vol. 112, pp. 800–818, 2020.

[8] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *IEEE Computer*, vol. 41, no. 7, pp. 33–38, 2008.

[9] J. D. McCalpin *et al.*, "Memory bandwidth and machine balance in current high performance computers," *IEEE computer society technical committee on computer architecture (TCCA) newsletter*, vol. 2, no. 19–25, 1995.

[10] L. W. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996*. USENIX Association, 1996, pp. 279–294.

[11] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller, "Memory performance and cache coherency effects on an intel nehalem multiprocessor system," in *PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 12-16 September 2009, Raleigh, North Carolina, USA*. IEEE Computer Society, 2009, pp. 261–270.

[12] S. Ramos and T. Hoefler, "Modeling communication in cache-coherent smp systems: a case-study with xeon phi," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*, 2013, pp. 97–108.

[13] J. Fang, H. J. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, "Test-driving intel xeon phi," in *ACM/SPEC International Conference on Performance Engineering, ICPE'14, Dublin, Ireland, March 22-26, 2014*, K. Lange, J. Murphy, W. Binder, and J. Merseguer, Eds. ACM, 2014, pp. 137–148.

[14] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems," in *42st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-42 2009), December 12-16, 2009, New York, New York, USA*, D. H. Albonesi, M. Martonosi, D. I. August, and J. F. Martínez, Eds. ACM, 2009, pp. 413–422.

[15] A. ARM, "Neon programmer's guide," 2013. [Online]. Available: https://developer.arm.com/documentation/den0018/a

[16] V. Babka and P. Tuma, "Investigating cache parameters of x86 family processors," in *Computer Performance Evaluation and Benchmarking, SPEC Benchmark Workshop 2009, Austin, TX, USA, January 25, 2009. Proceedings*, ser. Lecture Notes in Computer Science, D. R. Kaeli and K. Sachs, Eds., vol. 5419. Springer, 2009, pp. 77–96.

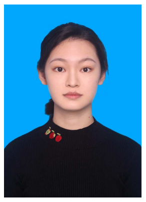
[17] H. Wong, M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos, "Demystifying gpu microarchitecture through microbenchmarking," in *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*, 2010, pp. 235–246.

[18] X. Mei and X. Chu, "Dissecting gpu memory hierarchy through microbenchmarking," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 72–86, 2017.

[19] L. James, Z. Xu, L. Cai, A. Nukada, and S. Matsuoka, "Evaluating the sw26010 many-core processor with a microbenchmark suite for performance optimizations," *Parallel Computing*, vol. 77, 06 2018.

[20] F. Mantovani, M. Garcia-Gasulla, J. Gracia, E. Stafford, F. Banchelli, M. Josep-Fabrego, J. Criado-Ledesma, and M. Nachtmann, "Performance and energy consumption of hpc workloads on a cluster based on arm thunderx2 cpu," *Future Generation Computer Systems*, vol. 112, 06 2020.

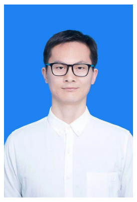
[21] S. McIntosh-Smith, J. Price, T. Deakin, and A. Poenaru, "Comparative benchmarking of the first generation of hpc-optimised arm processors on isambard," 2018.



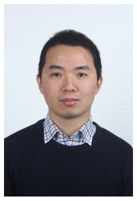
Wan-Rong Gao is a master student in computer science at National University of Defense Technology (NUDT). Her research interests are high-performance computing, system software, and performance optimization.

Jian-Bin Fang is an assistant professor in computer science at National University of Defense Technology (NUDT). He obtained his Ph.D. from Delft University of Technology in 2010. His research interests include parallel programming for many-cores, parallel compilers, performance modeling, and scalable algorithms. He is a member of CCF.

Chun Huang is a full processor in computer science at National University of Defense Technology (NUDT). Her research interests are high-performance computing, system software, parallel compilers, parallel programming, performance optimization, and high-performance math libraries.



Chuan-Fu Xu is an associate professor in computer science at National University of Defense Technology (NUDT). He obtained his Ph.D. from NUDT in 2011. His research interests include parallel computing and applications.



Zheng Wang is an associate professor at School of Computing at the University of Leeds. His research interests include compiler optimization, parallel programming and systems security.