Dissecting the Phytium 2000+ Memory Hierarchy via Microbenchmarking

Wanrong Gao, Jianbin Fang, Chuanfu Xu, and Chun Huang

College of Computer, National University of Defense Technology, China
{gaowanrong15, j.fang, xuchuanfu, chunhuang}@nudt.edu.cn

Abstract. An efficient use of the memory system on multi-cores is critical to improving data locality and achieving better program performance. But the hierarchical memory system with caches often works in a "black-box" manner, which automatically moves data across memory layers, and makes code optimization a daunting task. In this article, we dissect the memory system of the Phytium 2000+ many-core with micro-benchmarks. We measure the *latency* and *bandwidth* of moving cachelines across memory levels on a single core or two distinct cores. We design a set of micro-benchmarks by using the *pointer-chasing* method to measure latency, and using the *chunk-accessing* method to measure bandwidth. During measurement, we have to place the cacheline on the specified memory layer and set its initial consistency state. The experimental results on Phytium 2000+ provide a quantified form of its actual memory performance, and reveal undocumented performance data and micro-architectural details. To conclude, our work will provide quantitative guidelines for optimizing the Phytium 2000+ memory accesses.

Keywords: Phytium 2000+ · Memory hierarchy · Microbenchmark.

1 Introduction

Compared with single-core processors, multi-core processors have to deal with significantly more concurrent memory accesses [4]. The memory system has thus introduced a multi-level caching hierarchy to "lock" the frequently accessed data, aiming to minimize the accesses to the off-chip memory. Modern cache features, such as the number of cache layers, each layer's capacity, to use the *inclusive* or *exclusive* policy, and so on, vary across multi-core architectures. In addition, the memory system of modern multi-cores often works in the form of a "black box", i.e., many implementation details are not disclosed. And the official technical specifications only reveal theoretical numbers and is of little significance in guiding the actual performance engineering. All these bring programmers a huge challenge of optimizing codes on the cache-coherent multi-core architectures. Therefore, it is significant to dissect the working mechanism of multi-core memory systems through quantifying the actual performance behaviours.

Prior works have demonstrated how well the memory hierarchy performs on the conventional multi-core architectures. The STREAM benchmarks focus on

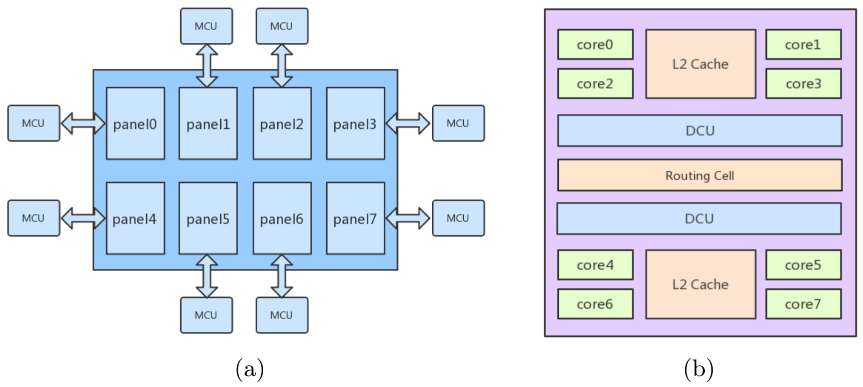


Fig. 1. The Mars II microarchitecture and the panel structure of `Phytium 2000+` .

measuring the memory throughput, i.e., data loading/storing bandwidth with a single-core or multi-cores [5]. McVoy and Staelin present a set of micro-benchmark suite (`lmbench`) to quantify the performance of various computer components [6]. In particular, they use *pointer-chasing* to measure the overhead of moving data across cache layers. But `lmbench` ignores the communication overhead of moving cachelines across processing cores. Such performance numbers are essential when optimizing parallel programs concerning shared memory accesses, producer-consumer or thread migration. For this, Molka *et al.* provide a set of microkernels (`BenchIT`) to characterize memory systems [7]. But `BenchIT` is only applicable to the `x86` architecture and its memory hierarchy.

In this work, we dissect the memory hierarchy and quantify the achievable performance on `Phytium 2000+` (an ARMv8-based cache-coherent 64-core architecture). We measure the communication performance of moving cachelines between distinct cores in terms of *latency* and *bandwidth*, through microbenchmarking (Section 3). We obtain undisclosed performance data and reveal many micro-architecture details of `Phytium 2000+` on both bandwidth (Section 4) and latency (Section 5). Our evaluation results provide a quantitative reference for analyzing, modelling, and optimizing the performance of parallel codes on multi-core processors. To the best of our knowledge, this is the first effort of dissecting the memory hierarchy of the `Phytium 2000+` architecture.

2 `Phytium 2000+` and Its Memory Hierarchy

`Phytium 2000+` uses the `Mars II` architecture [8]. Figure 1(a) gives a high-level view of the `Phytium 2000+` processor. It features 64 high-performance ARMv8 compatible processing cores. These cores are organized into 8 `panels`, where each panel connects a memory control unit (`MCU`).

The panel architecture is shown in Figure 1(b). 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. Note that, the L2 cache of `Phytium 2000+` uses a *inclusive* policy, i.e., the data cachelines stored in the L1 cache are also present in the L2 cache.

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, with a local interconnect on each panel and a global connect for the entire chip. The former couples cores and L2 cache slices as a local cluster, achieving a good data locality and short communication distance. The latter is implemented by a configurable cell-network to connect panels to gain a better scalability. `Phytium 2000+` uses a home-grown `Hawk` cache coherency protocol to implement a distributed directory-based global cache coherency across all panels.

3 Our Approach

This section introduces the design and implementation details of our benchmarks to measure the bandwidth and latency of the `Phytium 2000+` memory hierarchy.

3.1 Benchmarks Design

We measure the sustainable bandwidth by continuously accessing a chunk of data elements, which is shown in Figure 2(a). In contrast, we use *pointer-chasing* to measure the latency of loading a cacheline by randomly accessing discontinuous data elements (Figure 2(b)). In this way, we aim to mitigate the impact of hardware and/or software prefetching.

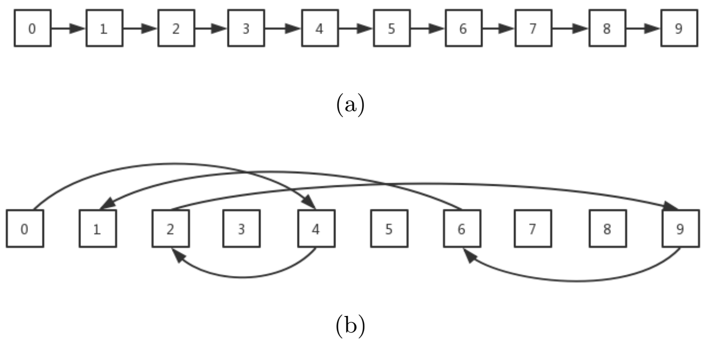


Fig. 2. The data accessing schemes for measuring bandwidth and latency: (a) accessing contiguous data elements to measure bandwidth, and (b) accessing randomly linked data elements to measure latency.

During the measurement, we use multiple threads to move data between cores. To ensure that the buffer allocated by a thread belongs to a fixed core, we pin each thread to a fixed core, i.e., thread n always runs on core n (`cn`).

Besides, we have to control coherency states (`modified`, `exclusive`, `shared`) of cachelines. We use the methods stated in [7], to set the initial coherency state. To determine which level the data is suited in, we control the size of the input datasets to be accessed. And we use a cache flush routine to replace the data in this cacheline with dummy data to evict the measurement data to the next-level cache. The benchmarking steps of measuring latency are shown in Algorithm 1. Here we assume that `c0` loads data from `cn`. The steps of measuring bandwidth are the same, except the way of preparing the initial data.

Algorithm 1 The benchmarking steps of measuring latency

```
Require: $n \geq 0$
 1: for $t = 1$ to $n$ do
 2:    initialize a thread $Thread_t$
 3:    cpu_set(mem_bind[t])
 4:    $thread[t].status \Leftarrow WAIT$
 5: end for
 6: for $t = 0$ to $n$ do
 7:    // Prepare data and set the initial coherency state of the cacheline
 8:    if $n == 0$ then
 9:       // Access local caches
10:       prepare_memory(thread[t])
11:    else
12:       // Access caches on other cores
13:       $thread[t].status \Leftarrow PREPARE\_MEMORY$
14:       prepare_memory(thread[t])
15:       $thread[t].status \Leftarrow DONE$
16:    end if
17:    // The cache flush routine
18:    if $n == 0$ then
19:       flush_caches(thread[t])
20:    else
21:       $thread[t].status \Leftarrow FLUSH$
22:       flush_caches(thread[t])
23:       $thread[t].status \Leftarrow DONE$
24:    end if
25:    // Measurement
26:    use assembly instruction to access data
27: end for
```

3.2 Benchmarks Implementation

When implementing benchmarks on the `Phytium 2000+` processor, we have to address the following architecture-specific details.

Enabling the clock-wise timing. Our benchmarks are designed to measure the performance of the Phytium 2000+ memory system. For such an measurement, we need a clock-wise timer. It is straightforward to do so by using the rdtsc instruction to read the timestamp on the x86 architecture. Similarly, 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. To address the issue, we use a kernel module to activate the performance monitoring unit. The key steps of this kernel module are summarized as follows.

– Reading the contents of the control register PMCR_EL0.
– Activating the user mode by writing PMUSERENR_EL0.
– Resetting all hardware counters by writing PMCR_EL0.
– Enabling the performance counter by writing PMCNTENSET_EL0.

With this kernel module, the PMCCNTR_EL0 register can be accessible through the mrs instruction to obtain the starting and ending timestamps.

Using the vector instructions. To obtain the maximum bandwidth, we have to 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 use of the same mnemonics as general registers [1]. The vector instructions are thus supported on the Phytium 2000+ processor. In the implementation of its SIMD instruction, registers can hold one or more elements of the same size and type. 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). When moving data between registers and memory, we use the LD1/ST1 instruction of the ARMv8 architecture, similar to movqda on the x86 architecture. The selected vector format is 4 single-precision floating-point words (.4S).

Using special instructions. Beside the general instructions, we use special instructions shown in Table 1. 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 space, we use DMB to ensure that the Phytium 2000+ processor does not optimize the execution order of the fetch instructions. In addition, we use the ALIGN instruction to avoid unaligned memory accesses.

Table 1. The special ARMv8 instructions [10].

DC CIVAC	Data or unified Cacheline Clean and Invalidate by VA to PoC
DMB	Data Memory Barrier acts as a memory barrier that explicitly enables the exeuction of memory access instructions in front of it.
ALIGN	Align instruction or data storage address

Table 2. `c0` read bandwidth (GB/s).

	Exclusive		Modified		Shared		RAM
	L1	L2	L1	L2	L1	L2	
`c0`	33.6	18.5	33.6	18.5	33.6	18.5	6
`c1`	13.3		13.3		18.5		
`c4`	10.5	10.9	10.5	10.9	10.9		
`c8`	9.2	9.7	9.2	9.7	9.3		5.1

4 Bandwidth Results

In this section, we measure the read bandwidth on the `Phytium 2000+` architecture. Figures 3 show the bandwidth of `c0` loading cachelines which are `exclusive`, `modified`, or `shared` in different cores and different cache levels. Table 2 gives a high-level view of the bandwidth numbers. We measure the bandwidth of `c0` loading data from its local cache, from `c1` sharing a L2 cache with `c0`, from `c4` on the same panel, and from `c8` on a different panel.

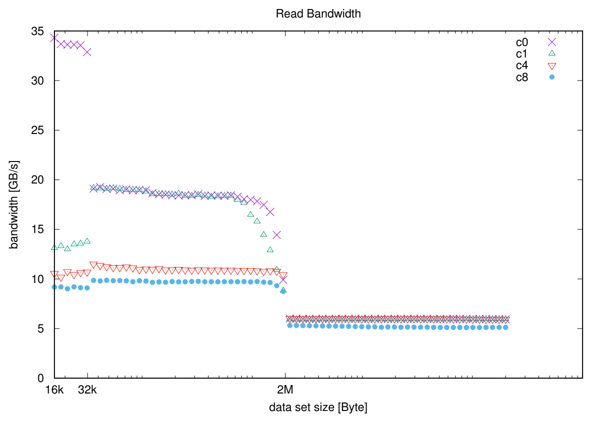
Local accesses. Whatever the state of the cachelines, data can be loaded from `c0's` local caches. The obtained bandwidth has nothing to do with the coherency state of the accessed data. The read bandwidth to its local L1 cache can reach 33.6 GB/s, while reading data from the local L2 cache can reach a bandwidth of 18.5 GB/s. Given that the L1 read port of `Phytium 2000+` is 128 bits in width and runs at 2.2 GHz, we calculate the theoretical L1 read bandwidth as

$$2.2 \times 128 \div 8 = 35.2GB/s \tag{1}$$

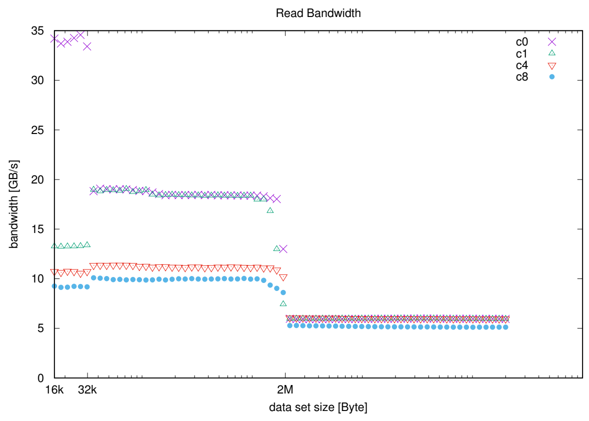
We see that the measured bandwidth is close to its theoretical counterpart (33.6 GB/s vs. 35.2 GB/s). The measured write bandwidth stays about 17.4 GB/s for L1. We note that the write bandwidth is around a half of the read bandwidth. This is because storing data into L1 occurs at 64 bits per cycle.

Within a core group. Given that `c1` and `c0` shares the same L2 cache slice, data can be loaded from the local L2 cache when the cacheline is `shared`. And the memory bandwidth of accessing the local L2 can reach 18.5 GB/s. The bandwidth stays the same when cachelines are `exclusive` or `modified` and suited only in the local L2. But the bandwidth is reduced to be around 13.3 GB/s when `c0` loading `exclusive` or `modified` cachelines suited in `c1's` local L1 cache. This is a notable difference from the `x86` processor that the data can be loaded directly from the shared cache slice, when the cacheline is `exclusive` initially. Only when the cacheline is `modified`, the data has to be loaded from the remote higher level cache. But on the `Phytium 2000+` processor, we observe that data have to be loaded from a higher cache level for both `exclusive` and `modified` cachelines.

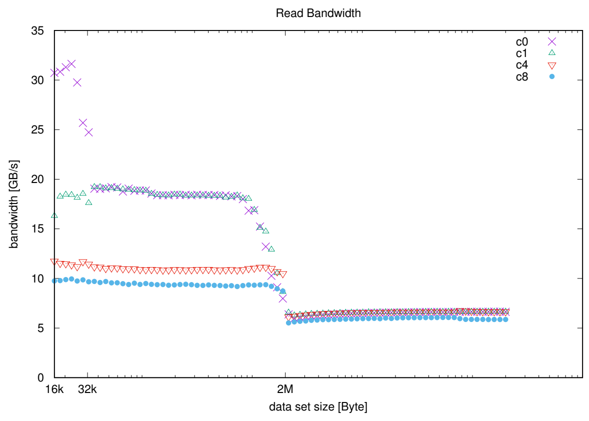
Within a panel. When `c0` loads data from `c4` of the same panel, where the two cores share no common cache slices, the bandwidth will be limited by cross-group links. As can be seen from Figure 3, the bandwidth is significantly smaller (by around 40%) than the case when sharing the same L2 cache slice.



(a) Exclusive



(b) Modified



(c) Shared

Fig. 3. Read bandwidth of `c0` accessing the local or another core (`c1`, `c4` or `c8`)

Table 3. `c0` read latency (cycle (ns)).

	L1	L2	RAM
`c0`	3(1.4)	21(9.5)	271(123.2)
`c1`	18(8.2)		
`c4`	34(15.5)	94(42.7)	
`c8`	49(22.3)	127(57.7)	310(141.4)

Similarly to `c1`, when performing cross-group access to `c4` for `exclusive` or `modified` cachelines, the bandwidth for reading the remote L1 cache is always smaller than accessing the remote L2. This is also because data can be obtained directly from the L2 cache only when its state is `shared` initially.

Across panels. `c8` does not share a common L2 cache slice with `c0`, and the two cores have to be communicated via the cross-panel routing cells. The read bandwidth of `c0` accessing `c8` ranges from 9.2 GB/s to 9.7 GB/s, which is smaller than the bandwidth of accessing `c1` or `c4` within a panel.

Memory. Since `c0`, `c1`, `c4` are within the same panel, they are connected directly to the same `MCU` and memory module. When accessing data in the local memory module for `c1` and `c4`, the bandwidth can reach around 6 GB/s. On the other hand, `c8` is connected directly to another `MCU` and memory module. The bandwidth of `c0` loading data from `c8's` memory module is around 5.1 GB/s.

To summarize, there is another difference between the `Phytium 2000+` processor and the `x86` processor when accessing the `shared` cachelines. The `x86` processor uses an extension of the MESIF protocol, which requires data to be fetched from the core with the latest copy (`forward`). Meanwhile, the `Phytium 2000+` processor uses a MOSEI-like coherency protocol. There is no need to find the `forward` copy, but it can directly obtain the data with an arbitrary `shared` copy.

5 Latency Results

This section shows the latency for the `Phytium 2000+` memory hierarchy. The performance numbers are measured when the cachelines are `modified` initially.

5.1 Overview of the Latency Results

Figure 4 shows the latency results when `c0` loading data from its local cache, from `c1` sharing a L2 cache slide with `c0`, from `c4` on the same panel, and from `c8` on a different panel. Table 3 shows an overview of the measured latency results.

We see that, the latency of accessing the local L1 and L2 cache are 3 cycles (1.4 ns) and 21 cycles (9.5 ns), respectively. For the `Mars II` architecture, there is no public specification documenting such numbers. The specification of the first generation `Mars` describes that accessing the local L1 and L2 takes 2 ns and 8 ns, respectively, which is in accordance with our measured results [11]. When `c0` loading data from `c1`, the latency is same as accessing the local L2 cache.

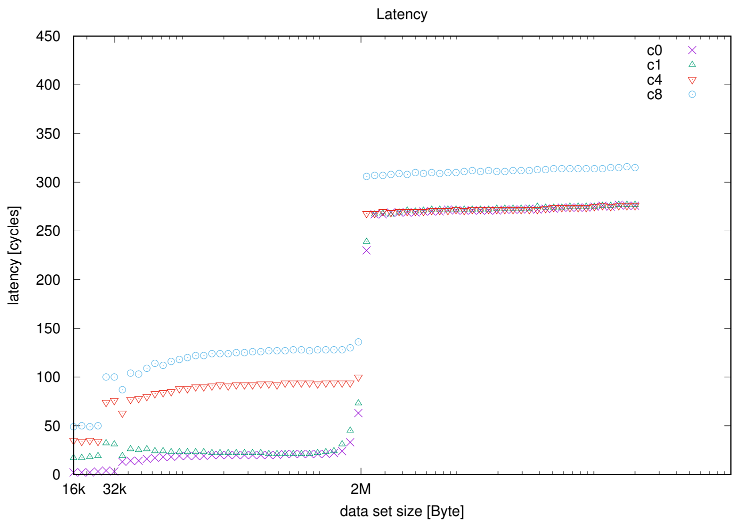


Fig. 4. Read latency of `c0` accessing the local (`c0`) or another core (`c1`, `c4` or `c8`).

Figure 4 shows that, no matter which memory layer the data is in, loading cachelines across core groups or panels takes many more cycles than accessing the local cache slices. Thus, loading data within a core group is the fastest.

5.2 Across-Panel Latency Results

We evaluate the performance impact of panel distance on latency when accessing cores fixed to different panels. Figure 5 shows the latency results when `c0` accessing the cores on `p1` (panle 1)–`p7` (panel 7), respectively.

We see that the latency numbers vary over the panel distance, with the latency difference of up to 105 cycles. Besides, the latency numbers of `c0` on `p0` accessing `c8` on `p1` and `c32` on `p4` are the same. It is the same for `c16` on `p2` and `c24` on `p3`. This is because these panels are at the same distance to `p0` (Figure 1). This result also agrees with the theoretical latency results [11].

5.3 With Different Page Sizes

We investigate the performance impact of the TLB page size on latency. Figure 6 shows the latency measured with the 4KB page, and the other configurations stay the same as that for Figure 4.

`Phytium 2000+` provides the usage of 4KB and 2MB pages. With 2MB page (Figure 4), the latency of each cache level looks stable. While using the 4KB pages (Figure 6), although the gap between different cache levels is still visible, the latency increases over the amount of data being accessed. This is because the latency measurement uses the pointer-chasing method. In the data preparation

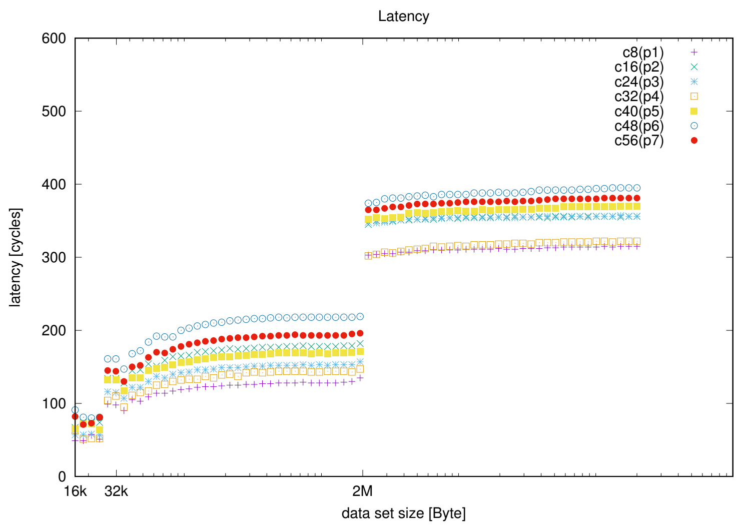


Fig. 5. Read latency of `c0` accessing `p1`–`p7`.

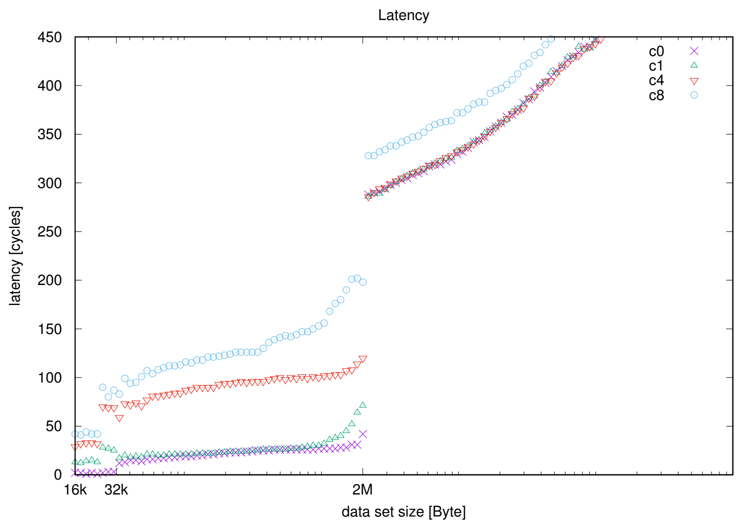


Fig. 6. Read latency of `c0` accessing the local (`c0`) or another core (`c1`, `c4` or `c8`) using 4KB pages.

stage, the next-to-be accessed address is randomly generated, resulting in a poor locality for the linked-list access. When using small pages, there will generate too many page table entries, leading to frequent TLB misses and resulting in a large memory access overhead. The bandwidth measurement does not have this issue because its access is consecutive.

6 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.* [2] 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 cacheline size, and the cache miss penalties.

McCalpin *et al.* [5] present four benchmark kernels (Copy, Scale, Add, and Triad), `STREAM`, to access memory bandwidth for a large variety 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.* [7] propose a set of benchmarks, including to study the performance details of the `Nehalem` architecture. Based on these benchmarks, they obtain undocumented performance data and architectural properties. This is the first work 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 [3]. Ramos *et al.* [9] propose a state-based modelling 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.

7 Conclusion

A variety of cache organizations and coherency protocols make modern multicores complicated, diverse, but hard-to-use. As the cache-based memory system is a critical factor that affects the overall performance, it is important to know its working mechanism and the achieved performance. This article focuses on dissecting the memory hierarchy of the `Phytium 2000+` architecture with microbenchmarks. Specifically, we quantify the on-core and core-to-core communication performance when cachelines are in different states and located in various cache levels. We choose `Phytium 2000+` as our experimental platform to access the performance of its memory system and dissect its working mechanism. The experimental results provide a detailed and quantitative performance description of the `Phytium 2000+` memory hierarchy. We also compare architectural properties between `Phytium 2000+` and the `x86` architecture. For future work, we will use the hardware counters in our micro-benchmarks to collect detailed performance data, aiming to obtain more details of the memory system, e.g., on the TLB miss rate.

References

1. ARM, A.: Neon programmers guide (2013)
2. Babka, V., Tuma, P.: Investigating cache parameters of x86 family processors. In: Kaeli, D.R., Sachs, K. (eds.) Computer Performance Evaluation and Benchmarking, SPEC Benchmark Workshop 2009, Austin, TX, USA, January 25, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5419, pp. 77–96. Springer (2009). https://doi.org/10.1007/978-3-540-93799-9_5, `https://doi.org/10.1007/978-3-540-93799-9\_5`
3. Fang, J., Sips, H.J., Zhang, L., Xu, C., Che, Y., Varbanescu, A.L.: Test-driving intel xeon phi. In: Lange, K., Murphy, J., Binder, W., Merseguer, J. (eds.) ACM/SPEC International Conference on Performance Engineering, ICPE'14, Dublin, Ireland, March 22-26, 2014. pp. 137–148. ACM (2014). https://doi.org/10.1145/2568088.2576799, `https://doi.org/10.1145/2568088.2576799`
4. Hill, M.D., Marty, M.R.: Amdahl's law in the multicore era. IEEE Computer **41**(7), 33–38 (2008). https://doi.org/10.1109/MC.2008.209, `https://doi.org/10.1109/MC.2008.209`
5. McCalpin, J.D., et al.: Memory bandwidth and machine balance in current high performance computers. IEEE computer society technical committee on computer architecture (TCCA) newsletter **2**(19–25) (1995)
6. McVoy, L.W., Staelin, C.: lmbench: Portable tools for performance analysis. In: Proceedings of the USENIX Annual Technical Conference, San Diego, California, USA, January 22-26, 1996. pp. 279–294. USENIX Association (1996)
7. Molka, D., Hackenberg, D., Schöne, R., Müller, M.S.: 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. pp. 261–270. IEEE Computer Society (2009). https://doi.org/10.1109/PACT.2009.22, `https://doi.org/10.1109/PACT.2009.22`
8. Phytium: Mars ii - microarchitectures. `https://en.wikichip.org/wiki/phytium/microarchitectures/mars_ii`
9. Ramos, S., Hoefler, T.: 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. pp. 97–108 (2013)
10. Rutland, M.: Stale data, or how we (mis-) manage modern caches (2016)
11. Zhang, C.: Mars: A 64-core armv8 processor. In: 2015 IEEE Hot Chips 27 Symposium (HCS). pp. 1–23. IEEE (2015)