On the exploration of the DRISC architecture

Yang, Q.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 4

The many-core challenge
—Problems of sharing

Abstract

The burgeoning many core systems not only challenge hardware researchers on the architecture side to grapple with emerging issues rarely seen before, but also press the software ecosystem to rethink or reorganize their work non-trivially, reconciling it with the coming revolutionary changes. Research in both the hardware and software fields faces the same issues and in spite of their distinct emphases, solutions of one domain are instrumental or complementary to the development of the other. Sharing is an example of this. This chapter will discuss the sharing issue from parallel programming to architecture design and general solutions in different domains. Meanwhile, the efforts toward conquering that problem in DRISC and Micro Grid research will also be examined.

Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Overview</td>
<td>60</td>
</tr>
<tr>
<td>4.2 Concurrent programming</td>
<td>60</td>
</tr>
<tr>
<td>4.3 Operating system</td>
<td>61</td>
</tr>
<tr>
<td>4.4 Resources on chip</td>
<td>63</td>
</tr>
<tr>
<td>Summary</td>
<td>73</td>
</tr>
</tbody>
</table>
4.1 Overview

Given the relentless quest for performance, the trend to integrate more and more cores onto a single chip is irreversible. The expected number of cores on chip invalidates the current duplicate and apply policy, as in the shift from uni-core to dual-, quad- or octo-core processors and gives rise to a scalability requirement.

For application developers, scalability indicates how to fully exploit the power of the provided computing resources so as to achieve flexible performance on various target platforms without insurmountable ceilings.

For system software developers, scalability chiefly means how to interact with both upper level user applications and lower level hardware entities efficiently so as to achieve transparent application performance without additional overheads and refactoring.

For hardware developers, scalability by and large denotes how to design a reasonable architecture to stay compatible with future evolution while maintaining sound performance efficiency without a thorough rebuilding.

Realizing scalability challenges both software and hardware designers because of different and diverse concrete issues associated with many-core systems. However, they all have to crack the same nut—sharing, which is the subject of this chapter. It will touch on the software domains and focus on the architecture field closely relating to our research.

4.2 Concurrent programming

The “free lunch” that let the processor itself exploit parallelism from binary code implicitly has already gone, and scalable performance demands that concurrency is explicitly exposed to hardware in such a way that the task is partitioned into subtasks (threads), which then can be dispatched to multiple cores to achieve acceleration. Agglomeration should be meticulously tuned as it may be too coarse under-utilizing existing computation power or too fine causing parallel overheads that counter any expected benefit. A suitable value usually is target dependent and has to consider communication and synchronization costs most of which are because of inter-dependencies from shared data.

Languages adopting the shared memory model such as OpenMP and Cilk rely on locking (mutex or semaphores) to guard around shared data so that they can only be modified by one thread once it has acquired the locks. Meanwhile other threads have to spin there waiting for a free lock to continue execution. The locking mechanism is valid to eliminate data races and ensure mutual exclusion but is costly, let alone that the locks themselves are shared leading to side effects. Alternatively lock-free mechanism are widely studied to avoid these negatives. Lock freedom is accomplished through atomic primitives like read-modify-write and compare-and-swap, or with transactional memory which has better abstractions [HLR10, FC11] or basic data structures, e.g. arrays [DPS06], queues [ST05], lists [TBKP12] and trees [BH11]. Unidirectional communications on shareds and exclusive places of our programming model can also be regarded as a choice to replace locking.

The situation is better in the message passing model leveraged by, for example, Erlang, Charm++ and occam because of “sharing nothing”. All interactions are
4.3. OPERATING SYSTEM

made via messaging synchronously or asynchronously and code can be flexibly deployed to hardware platforms even with shared memory. The major concern comes from separated data use and storage. References to non-local data must be fetched from their hosts first and used later. This causes storage overheads for data copies and also the overheads of data marshalling. Additionally raised problems of message passing are the reliability of communication channels and traffic exposed to the interconnections. The former is less pressing to many-core systems with on-chip connections in contrast to HPC-oriented cluster systems with extraordinarily long wires, but the latter challenges the scalability of many-core systems and we will investigate this in section 4.4.3.

No matter which model is taken, user applications depend on an OS to execute on hardware platforms and their performance is more or less affected by the service efficiency of the OS.

4.3 Operating system

Operating systems act as a layer between the user applications and hardware, and are in charge of assisting and serving software to run efficiently on target platforms and attain the deserved performance. Therefore they themselves should have negligible overheads to ensure performance transparency as far as possible, and this demands good scalability of the OS along with the evolution of many-core systems. “The irony is that hardware is now changing faster than software, and the effort required to evolve such operating systems to perform well on new hardware is becoming prohibitive [BPS+09].” It is the uncontrolled sharing of resources and the use of shared memory to manage kernel data structures, for instance tables of file descriptors and process IDs, which are often shared by all cores by default, that hinder the scalability of legacy operating systems [KTR+12].

The monolithic OS in the uniprocessor era has no other way but to share hardware resources with user applications; thus mutual interference is inevitable such as multiplexed pipeline, frequent cache and Translation Lookaside Buffer (TLB) misses, etc. The OS kernel also applies a global lock to all sharing-sensitive kernel states to guarantee exclusive accesses from concurrent threads. The prevalence of multi-core processors have only recently impacted OS design in a way that global locks are replaced by fine-grained locking. This moderates concurrency obstructions yet keeps the resource sharing paradigm almost intact. Such a revamp might be useful for fewer cores but will be stretched soon in front of the rapid quantitative and qualitative evolution of many-core processors far outpacing the rate at which OS developers are capable of redesigning subsystems [WA09]. This propels development in breaking the sharing bottleneck and bearing the fruit of scalable many-core OS.

Corey [BWCC+08] attacks the data sharing penalties by letting share-ability of kernel data structures be controlled and directed by applications. Each system data structure is initialized private to one core instead of being shared by all cores by default and applications running on a core can make decisions to share private data belonging to that core with some others. Note that “applications” are in the general sense of ordinary user applications, application-level libraries and even OS services. To reduce frequent sharing requests and maintenance, Corey further proposes the abstraction of kernel cores dedicated by applications to
perform specific functions. Application cores could outsource network transactions to a web service core by simply contacting and sharing data with it in order not to manipulate and contend for driver and device data structures.

Tessellation \[LKB^+09\] manages resource sharing by introducing space-time partitioning. On-chip resources such as cores, shared caches, network bandwidth and even energy budgets are dynamically divided into partitions assuring exclusiveness or Quality of Service (QoS) and these virtual spatial partitions are scheduled globally at run time onto available hardware as a whole in a time-multiplexing way. Thus assigning user applications and OS services to different partitions could diminish mutual interference. Partitions communicate with each other only through reliable messaging for intra-application activities (one application across several partitions) or interactions between user applications and OS services. Consequently there are flexibilities in that partitions can be activated by incoming messages or conducted by user-level scheduling for specific purposes.

FOS \[WA09\] and Barrelfish \[BBD^+09\] make breakthroughs in OS architectures. They do share something in common but envisage many-core OS differently. FOS decomposes the OS into a set of services each of which is on par with a distributed internet server consisting of multiple server processes and can be spatially distributed across the entire chip. Each server is bound with a specific core, communicates and collaborates with others to act as an integrated OS. Moreover distribution, as well, extends to services and data-structures deep in the OS kernels. Applications occupying the remaining cores resemble web clients and messaging servers for kernel data structures and services. FOS-micro kernels as the lowest layer are deployed to each core on chip to assist communication, access control etc. In a nutshell, FOS converts the sharing problems into layout and partition problems; and the scalability challenge now becomes one of determining optimal placement of processes for good performance.

Barrelfish treats many-core chips as a network of independent cores and views the OS architecture from the perspective of distributed systems, assuming no inter-core sharing at the lowest level. This yields the proposed OS architecture for heterogeneous many-core systems under the principles that all cores have to communicate via message passing explicitly, all kernel data structures have to be replicated if shared and the OS structure except the interface to hardware should be target-neutral. Apparently the first two restrictions are to cast off the sharing burden and the last is for the sake of heterogeneity. However replication incurs the overhead of consistency maintenance namely updating state via exchanging messages. Barrelfish does not favor spatial isolation arguing that enough concurrency may exhaust on-chip resources (chiefly due to the factor of heterogeneity) and varying execution demands undermine QoS provision \[PSB^10\].

Switching to our research on the DRISC many-core architecture, there is no independent OS kernel till now but it is rather dispersed within the Micro Grid simulator. Partial services and functions have been defined and packaged into libraries analogous to libOS \[EK^+95\], in addition to that there are basic services realized in silicon (i.e. concurrency management as well as communication according to register types). On top of the homogenous platform, each OS service is statically bound with a dedicated exclusive place during system initialization. The references (e.g. procedural calls) to these services from user applications will be converted to remote family creation on corresponding places. Taking previous
web transaction as example, the procedural call of web service function on a core will be transformed into sending a synchronous or asynchronous creation request to the target place. Then a singleton family will be created at this remote place to process the transaction and the related data is communicated via shared memory. The property of exclusive place ensures all incoming requests will be queued first and processed one after another by means of FCFS. In this way mutual exclusion on shared data structures is guaranteed and core affinity is exploited. Despite Barrelish’s concerns of space sharing, family resources of applications are dynamically allocated; resource contracting and auto-compromising execution promise run-ability even though there may not be sufficient resources; of course this is at the cost of performance. [van13] has carried out extensive studies on the OS design of DRISC many-core architecture and readers can move there to get the panorama of the research.

No matter how OS and user applications are specially arranged, once executing there is generally no difference between them as far as underlying many-core platforms are concerned, and the fact that they temporally share or concurrently compete for the same resources is immutable. It is from this situation that problems result as part of the scalability issue studied by architecture designers.

4.4 Resources on chip

Multiple cores on chip facilitate space sharing so that applications (also in the general sense) obtain primarily exclusive computation resources, namely those within the domain of an individual core, but this does not and will not eliminate the sharing of NoC, memory and other precious resources like the shared FPU. With multithreading and multiple cores, threads always compete with each other not only in the same time-multiplexed core but also with others (derived from the same or different tasks) executed concurrently in any other different core. In all cases inter- or intra-task interference is inevitable, resulting in the upshot that firstly one application’s performance is variable and dependent on different companions, and secondly the system scaling yields unmatched or worse performance efficiency even when appreciable parallelism is leveraged. These problems comprise the scalability issue of many-core architectures, some of which could be resolved or mitigated by hardware improvements and some of which rely on or could be complemented by software optimizations. We will scrutinize the sharing of memory and on-chip interconnection and corresponding solutions in the following subsections.

4.4.1 Shared on-chip memory: hierarchical caches

The introduction of caches is to fill the speed gap between cores and off-chip memory so that a memory access will not degrade performance too much. Architectures with hierarchical caches dominate the current landscape\(^1\) and are still likely to be popular in future many-core systems.

\(^1\)There are exceptions such as Cray’s XMT and Adapteva’s Epiphany with no caches.
4.4.1.1 Locality issue

Caches can feed cores with requested data as soon as possible if locality is well maintained however sharing caches among threads (private caches for cores) or cores (Last Level Cache (LLC)) usually hurts locality and cache misses do harm performance.

One possible solution to reduce cache misses is to increase cache size so that all needed data can be always hold there without eviction. The key is what a suitable capacity is. IBM’s POWER7+ is equipped with 80MB LLC [SKS+11] from which each core gets 10MB and each thread gets 2.5MB on average. Tilera also provides 18MB LLC on its 72-core chip [Ada12] and each core obtains only 0.25MB. Theoretically the larger capacity the better result but cost is the primary concern e.g. the 8KB private cache (covering both I-Cache and D-Cache) consumes 40.28% of a DRISC core as well having potential issues with the access latency. Although the innovative eDRAM technology consumes two thirds less area than SRAM does and possesses superiorities in latency and standby power [SKS+11], it is still insufficient to support all on-chip cores given the large quantity envisaged in the future, the finite chip size budget and diverse application demands. While cache compression affords an alternative to virtually expand capacity without adding chip area occupation [YKK04]. Academia makes extensive studies on compression and decompression algorithms like [LHK00, PA05, CYD+10, PSM+12] for a long time to ensure less latency and energy overheads.

Other than methods with respect to capacity, improving locality can also resort to providing efficient management. Research on this theme by and large runs in two directions, namely promoting constructive sharing and reducing destructive interference. The former focuses on scheduling cooperating threads which share overlapped data set. For instance, [AC06] investigates co-scheduling cooperative threads in the real-time field and achieves sizable L2 miss ratio reduction; [CGK+07] explores two scheduling algorithms and their effectiveness with different cache parameters; [JTS10] piggybacks on program locality analysis to predict online job co-scheduling while [ROA13] reduces cache footprint by dynamic thread throttling based on the feedback from memory system. The latter stresses fair cache sharing and partitioning like what has been done in [KCS04, XL09, KIJ13]. [HGC10] proposes elastic cache hierarchies to make caches either private or shared automatically and autonomously to application behavior. Note that some of above techniques also rely on OS efforts to function better.

[ZJS10], after investigating application performance in PARSEC [BKSL08], argues that the advantages of cache sharing management can only be at full play if programs are transformed to improve shared cache usage. This may contradict a large body of work in OS and architecture design; however it conversely manifests that programming or even compiling is prodded into action to make better use of existing optimizations or to alleviate sharing. [KT02] and [ST08] have made efforts on compiling to change instruction and data placement in order to reduce cache conflicts.

On account of chip size assumptions, Micro Grid affords rather small caches, namely 4KB private data cache and 128KB L2 cache as recommendations. Although private cache accesses are already on critical paths, currently there is no special cache management or optimization as mentioned above except for that
places are often made application exclusive and correlated threads are deployed to share overlapped data. Even so locality is vital to system throughput. Figure 4.1 shows the locality impact from either data redistribution or together with larger caches on top of two encryption kernels that hold large data sets. Note that these applications execute multiple streams per core, one stream per thread and hence as the number of threads increases, the impact of cache sharing is exacerbated. Figure 4.1a illustrates the importance of locating input data in the working core that refers to it and Figure 4.1b the additional benefits from increasing cache size. It is out of question that larger caches achieve better result; however if taken hardware costs into consideration, the recommended cache configuration has better trade-off between performance and chip size. This is proved by the system throughput yield per area in fig. 4.2, which also shows it would be unwise to resort to cache capacity only for system performance improvements in a many-core chip with massive concurrency. Anyhow, reasonably improved performance depends on both mandated data sizes and concurrency scale producing different cache footprints, yet part of it is attributed to cutting coherence overheads.

4.4.1.2 Coherence issue

Unlike purely message passing systems, maintaining data consistency across caches is the essential prerequisite for correct execution on those systems with shared memory, which can be accomplished either by software or by hardware. Although shared memory systems with hardware coherence are prevailing at present, there is no consensus whether this is sustainable in the many-core future.

Some are pessimistic about the scalability of shared-memory abstraction as support for hardware coherence, and suggest switching to software concurrency, explicitly managed scratchpad memories or message passing [MHS12]. For example, FOS and Barrelfish in section 4.3 have shown such concerns and re-base their work on top of message passing for forward compatibility. The 48-core SCC [MRL10] uses hybrid memory models with split memory spaces and enforces message passing for inter-core or chip-wide communications, which is supported by dedicated on-chip message buffers and the communication environment RCCE; thus conducting coherence is in the charge of programming and is implemented over message passing.

It is the ease of programming and management, as well as portability issues that spur research to promote hardware coherent shared memory to even more cores. [MHS12] argues that overheads of on-chip cache coherence could scale gracefully with core count and are similar to that deemed acceptable in today's systems. To refute “conventional wisdom”, it explores scalability issues and concludes that accurate tracking of sharing blocks and hierarchical inclusive caching with better encoding are efficient mechanisms to control traffic and storage costs, on the basis of which the latency and needed energy grow only with the number of cores. Also with directory-based cache coherence, [MGT99] proposes sparse directories, one entry of which tracks several actively shared memory blocks coupled with coarse vectors tracing the exact sharing information of each block to lower storage costs. WAYPOINT in [KJLP10] amalgamates sharing information benefits of ideal full directories and the storage benefits of sparse directories. It manages to scale to one thousand cores through dynamic capacity and associativity extension of
(a) Locality-aware data partition in coding with default cache configuration.

(b) Larger cache supplements on the basis of well located data. The 64KB L1 configuration requires a 512KB shared L2 cache.

Figure 4.1: Locality impacts on system performance measured by average IPC per core. RC4 and MD5 are running on the 128-core Micro Grid chip with DCR. Each core runs 1 ∼ 16 threads and the platform follows the configuration in table 3.1 (c.f. section 5.5.2 and table 5.3 for benchmark details.).
4.4. RESOURCES ON CHIP

Figure 4.2: Cost efficiency of each cache configuration. This is calculated as the IPC yield per unit of chip area. Unmarked grey lines are the maximal unitary values of corresponding cache configurations assuming all on-chip cores are always 100% busy.

directories as well as novel entry eviction handling. Other relevant studies include refined work on bloom filters in [ZSDS11, FLKBF11, SK12] for the interests of lower chip area and energy costs.

Besides the classic bus or directory protocols, scalability is also approached by designing novel protocols. [ZLS10] proposes the performance critical treefractal coherence protocol for many-core systems that are logically organized as trees. The fractal behavior and coherence of a scale are maintained by its own fractal interface and all hierarchical interfaces of the entire tree assure system-wide coherence synergistically. The treefractal approach gains an advantage in storage cost but fails a bit in access latency and energy because of greater tag associativity. [ZFJ+13, YJ13] leverage token coherence, which was developed one decade ago in e.g. [MHW03a, MHW03b] as an alternative to get rid of the ordering penalty in conventional methods, to construct new protocols. The former settles the traffic issue of token coherence with a sub-net so as to scale to many cores; the latter accomplishes the DP&TB coherence protocol at the granularity of a page by employing both directories to detect sharing of pages and tokens to maintain coherence within a shared page.

DCR of Micro Grid (c.f. section 3.4.2 and fig. 3.11) also guarantees hardware coherence and does this hierarchically with hybrid protocols. The coherence within a L2 cache and its clients is ensured by a snooping protocol; coherence amongst L2 caches in the ring relies on a write-update protocol and the sharing state relies on tokens. This avoids bulky directories and sharing tracing but incurs other problems. We will elaborate the details and the consequences of this in chapter 5.
4.4.1.3 Consistency issue

Other than cache coherence which concerns what value can be returned for a read, memory consistency is about when a written value from one core can be read or must be seen by the others, or generally “what properties must be enforced among reads and writes to different locations by different processors [HP12]”. This is another issue that must be addressed in shared memories.

Sequential consistency imposes exact memory access order arbitrarily and is the most straightforward and simple for programming; however it is too strict to exploit compiling and architecture optimizations rearranging instruction execution order and is thus performance expensive. Because of this, relaxed consistency models that allow relaxed memory access orders prevail in current main-stream architectures but they usually differ in exact degree, for instance the most aggressive model of the Alpha compared with the more rigid models of IBM’s zSeries and AMD64 [McK10]. Meanwhile each provisions special primitives to enforce memory order when necessary but with mandatory inter-core or even chip-wide synchronization. This brings extra latency and another factor that impedes scalability. Recently [NJL13] analyzes the scalability of several state-of-the-art memory consistency models mentioned in [McK10] on a 64-core platform of distributed shared memory with 2-D mesh interconnect. It observes relaxed models yield rather good performance over sequential consistency and expects more performance differential when increasing the network size. Nevertheless these absolute results do not say anything about the scalability for each relaxed model individually and it is still important to lower the cost.

Optimizations to alleviate memory ordering penalties usually fall into two categories, i.e. reducing either the number of ordering operations in instruction streams or the synchronization cost of each operation. [VPCCR06] claims current processor-centric ordering techniques via locking often cause superfluous memory ordering operations and are thus costly. It proposes conditional ordering, allowing cores to interact with each other to activate ordering imperatives according to historical events. This ordering scheme is implemented by introducing a new programming model and a release counter in hardware, and shows substantial performance benefits across Java benchmarks only if there are fewer remote synchronizations with reasonable latency. [LNG12] does similar work to perform only the necessary stalling dynamically with the help of compiling information and [SMSW09] on software transactional memory from compiler optimization. Partitioning instructions into different chunks and maintaining ordering at the granularity of a chunk instead of an individual instruction provides another choice to decrease ordering frequency [CTMT07]. Speculation is widely studied to moderate the runtime synchronization penalty [WAFM07, BMW09] but it has to take care of storage cost and recovery to minimize the negative effects. [AFA12] develops hardware barriers and a dedicated network to enable fast and efficient synchronization processing, while [SK10] investigates how to reuse existing interconnections on chip to increase chip-wide synchronization performance to be comparable with a dedicated synchronization network.

We have articulated our memory consistency model in section 2.2.6 and the counter-based implementation in Micro Grid (c.f. section 3.3.2). The relaxed consistency requirement that is restricted to the parent thread and its children
firstly dismisses additional inter-core synchronization from ordering imperatives. It can even be integrated into family bulk synchronization and amortized by thread alternation; and secondly it results in no stalls as long as there are sufficient active threads. Its performance impact is determined by the memory subsystem or more precisely the cache coherence protocol. We will explore this in section 5.4.3.

4.4.2 Shared off-chip memory: pin bandwidth

The performance discrepancy between processors and external memories touched off the “memory wall” and triggered a turning in architecture design to tolerate memory access latency. This brought about hierarchical caches, multithreading, etc. The lagging evolution of memory versus processor performance thrusts memory again into the spotlight as it is often “a fundamental performance and power bottleneck in all computing systems [Mut11]”. The dilemma is caused by, on the one hand, a rapid increase of on-chip core count and soaring diverse data demands from emerging applications [LCD11] exact higher bandwidth requirements; on the other hand the number of memory controllers is constrained by pin-limitations, packaging and energy costs, and the Dynamic Random Access Memory (DRAM) frequency and capacity are difficult to scale up via smaller nodes [LIMB09]. These give rise to the problem of the “bandwidth wall” at the chip boundary. [RKB+09]’s analytic model shows if this does not change, the bandwidth issue will slow down the scaling of many-core chips by a factor of 5.

[MFP10] demonstrates that the exposed memory concurrency is the fundamental determinant to performance as concurrent accesses sharing the same hardware resources make it easy to saturate the different levels of memory and to stop any increase of bandwidth. To mitigate this, [CLLY10] proposes phased thread execution throttling based on previous computation to memory ratio while [ELMP10] does so by applying a global threshold to all shared resources in the memory system. [KHMHB10] resorts to prioritizing and scheduling threads that require the fewest services from memory controllers.

Given the role of hierarchial caches on chip, strategies such as those mentioned in section 4.4.1.1 for improving locality to make full use of caches are equivalent to reducing off-chip traffic and alleviating the pin bandwidth pressure. [BGK96] concludes that complex on-chip caches are cost-effective in tackling the pin bandwidth limitation. [RKB*09] also proves a combination of several advanced cache techniques has the potential to enable super-proportional core scaling for up to 4 technology generations within the bandwidth envelope and this requires up to 71% chip area for caches.

Unless peak bandwidth is demanded and achieved at any time, making full use of the provided bandwidth to serve all memory accesses as far as possible is also a kind of amelioration. This partially stresses the interference between reads and writes. Due to the DRAM nature, frequent switches between writes and reads impose significant data bus idle time and force reads to be queued longer at memory controllers. Thus available bandwidth is dampened. [LNE*10] tries to conquer this problem by reducing write frequency and improving write locality in the DRAM row buffer. It proactively retires dirty blocks in LLC in addition to the Least Recently Used (LRU) if they will hit the same row of DRAM. All writes will be committed upon a full write buffer. Differently [CMB+12] manages
Figure 4.3: Investigate pin bandwidth limitation on the Micro Grid using random DDR memory (fig. 3.10) configured as in table 3.1. This is made by running Grey on 16 and 128 cores with $1 \sim 64$ threads per core and measured by average IPC per core. The number of DDR controllers is set to 2, 4 or 8, each of which offers 102.4 GB/s peak bandwidth (c.f. section 5.5.2 and table 5.3 for benchmark details.).

to improve read-write parallelism by splitting reads into two stages and performing pre-fetches in concert with writes. It relies on additional registers attached to I/O pads to buffer the pre-fetched data.

Since Micro Grid leverages multithreading to tolerate memory access latency, system throughput can continue scaling up along with increasing parallelism theoretically, however this is not always available in practice as reflected in fig. 4.3. Given the rather small private caches, both the quantity and write to read ratio of off-chip memory accesses are almost invariants in all contexts (fig. 5.14a shows the 128-core situation.). The disparities of achieved average efficiency between 16 cores and 128 cores reaffirm memory concurrency impacts due to fierce contentions for memory controllers. Increasing concurrency is able to improve bandwidth utilization but this is likely to saturate shared memory controllers and stop the benefit sooner with many cores. Increasing off-chip channels is useful to achieve higher bandwidth (or system throughput) yet the effect varies and depends on how much it betters the “many to few” paradigm.

Thanks to the addresses striping across memory controller in the cache line size and the nature of this random DDR memory, the imbalanced workload distribution amongst memory channels does not appear here, which is argued in [MG11] as an important factor for performance degradation. But it does happen to DCR. Besides the chip boundary limitation, the earlier performance saturation with concurrency scaling can also result from on-chip interconnection.
4.4.3 Shared interconnection: on-chip bandwidth

Accompanying the development of processors, interconnection also evolves to meet the need of chip scaling, e.g. from simple bus to crossbar. Nowadays these conventional structures are difficult to afford highly integrated chips with numerous and diverse components. A single chip more and more resembles a small scale network system and all components should be connected in a flexible and scalable way backing high performance. The NoC is recognized as the candidate by common consent, which after the tenet of “rout packets, no wires [DT01]” can piggyback on mature networking theory and methods and put them into on-chip communication. After all, a chip is not a network system and NoC has to have specific architectures for various demands, one critical cross-cutting design challenge of which is latency [ODH+07]. Note that reducing latency is equal to improving available bandwidth under certain assumptions.

Provided all memory transactions and inter-core communications would be payloads of the NoC, the aforementioned cache optimizations, shared memory aware scheduling and redundant memory ordering removal diminish either the amount of traffic or the concurrent hits, and thus are helpful for better on-chip bandwidth utilization and should be orthogonal to any NoC optimization.

Apparently, the latency or bandwidth issue can be easily resolved by building dedicated connections for different on-chip transactions. This is exemplified by Tilera’s iMesh network offering five independent connections, one of which is for scalar communication between cores, two dynamically routed networks are devoted to memory subsystem and the other two for application and I/O use. iMesh gets rid of all virtual channels, enables one cycle latency per hop and affords 240GB/s bisection bandwidth in total. This of course should be covered within the area and energy envelope. As a representative of optical approaches instead of electronic-based interconnects, [KL04] investigates building a photonic NoC for shared memory systems. It manages to maximize available communication channels on top of physical properties of light, namely decentralized wavelength channel allocation, time, space and wavelength division multiplexing, and to achieve lower price and better scalability, but the conversion between photonic and electronic signals is usually the “last mile”.

“Orthodox” research yields ramifications on solutions. [DMN+08] proposes both storage and communication compression, the NoC strengthened work over cache compression, and tries to pipeline compression, communication and decompression to hide any extra latency. [NFM+12] studies the congestion problem in buffer-less NoC and adopts an application-aware throttling algorithm to dynamically monitor the ratio between retired instructions and created traffic and then throttle applications of lower ratio. Actually traffic jam alleviation is a traditional topic and a criterion of router design. This and latency requirements challenge routing mechanisms that should be simpler and more efficient than traditional networks. Deterministic mechanisms such as dimension-ordered routing are well-known for simplicity but they lose flexibility to adaptive strategies in which a router makes decisions upon dynamic information collected from its neighbors or larger network areas. For example, C-Routing [PSGL11] possesses partial adaptivity enabling three of four routing directions to be neighbor-awareness for the interests of smaller routing table size and virtual channel freedom. [GGK08] divides the whole NoC into regions and monitors status of neighboring regions to
CHAPTER 4. THE MANY-CORE CHALLENGE

Figure 4.4: Investigate on-chip bandwidth limitation in Micro Grid with DCR (fig. 3.11) configured as in table 3.1. This is made by running Grey on 16 and 128 cores with 1 ∼ 64 threads per core and measured by average IPC per core. The peak bandwidth of memory subsystem ring is set to 64(lb), 76.8(mb) and 89.6(hb) GB/s (c.f. section 5.5.2 and table 5.3 for benchmark details.).

improve information coverage thus approximating global inherency. [MJW11] attempts to offer as much information as possible for routing decisions while taking into account interference among applications in case of abundance. There are also techniques to switch between these two kinds of mechanisms dynamically based on the network congestion situation as [HM04] explains, to exploit the advantages of both.

Micro Grid assumes three types of connections for different purposes, namely inter-core connections for place allocation and concurrency creation and synchronization (c.f. section 3.3.1), connections of I/O subsystem (c.f. [HvTJ10, van13]) and memory subsystem (c.f. section 3.3.2). The physical implementation for the first type is not presumed and is open to any feasible scheme, but a dedicated link might be unnecessary due to less frequent usage. The second is out of the scope of this thesis and we focus on the third as a part of scalable memory system. Currently we have a built DCR, using a simpler topology than frequently studied mesh. The simplicity of the ring in both topology and routing allows for very wide data paths and hence a high link bandwidth. The DCR is accurately simulated within the Micro Grid simulator and helps capture bandwidth problems shown in fig. 4.4. As the maximum bandwidth of the ring, under non-local traffic conditions, is less than that of a single channel, there is no benefit from extra off-chip channels. What is the same with fig. 4.3 is the concurrency required for saturation, while the better results highlight the effectiveness of L2 caches. It is notable that large concurrency scale is in favor of higher speed connection. The performance benefit of 128 cores is in an average of 16%, nearly the same as the 20% peak bandwidth
increment. This on the other hand indicates the on-chip bandwidth limitation. We just present results here and leave the reasons and solutions to chapter 5.

Summary

To achieve a many-core system with appreciable scalability in performance, sharing problems must be conquered or contained rationally. Work on the basis of message passing may be less pressured due to the loose dependency on implicit sharing while a lot of efforts are also devoted to pushing shared memory toward better scalability. Disregarding their differences, on-chip resource sharing is a common issue and is often uncircumvented, which often gives rise to inter- or intra-application interference reflected by dependent or worse scaling in performance. Pin and NoC bandwidth are two examples. This is usually mitigated by quantitative or qualitative improvements of the component itself, the optimized task scheduling in concert with both hardware and OS or/and the source control from programming and compiling.

The research of DRISC and the derived many-core system also addresses the sharing problem in different aspects. We introduced specific resolutions within the software domain and have studied the resource sharing influences on system performance when exploiting multiple threads and multiple cores. The explored solutions from literature are good references for our continuing research and we also would like to investigate more suitable schemes for DRISC. In the next chapter is an example of dealing with bandwidth problem through optimizing the DRISC architecture.