Cache Coherence Scaling on Manycore Systems

Oliver Türk
Friedrich-Alexander-Universität Erlangen-Nürnberg
oliver.tuerk@fau.de

ABSTRACT

On-Chip cache coherence is in widespread use on mainstream general-purpose computers nowadays. Scaling from multi to many core systems hardware coherent cache might become problematic, since simply scaling existing systems is not possible due to overhead. This paper will discuss and evaluate two different approaches for cache coherence implementations in many core systems, that solve the overhead problems caused by a high count of cores. One of these approaches stays with hardware cache coherence but introduces a hierarchy. The other is based on a hybrid approach, based on hardware cache coherence on small scale and software cache coherence on coarse scale.

1. INTRODUCTION

Current general-purpose processors build in x86 or amd64 architecture utilize only few cores - between 4 to 8 in consumer products and up to 24 for professional server machines.Since Pollack's Rule states that the performance of a microprocessor grows roughly proportional to the square root of it's complexity, it is not favorable to stay with the same core count. If inverted Pollack's rule gives implies a speedup of factor $\sqrt{2}$ when dividing a core into two. Therefore the logical solution is a increasing core count, which brings issues with currently used cache coherence designs.

Hardware cache coherence is the widespread design for current processors on which all mainstream software is build upon. The second design in use are graphics-processors (GPUs), already using thousands of cores on one chip [1]. GPUs are not using coherent caches though [10], making programming more demanding. Therefore a GPU-like design will probably not be the solution for mainstream processors but a hybrid cache coherence approach, utilizing hardware cache coherence on small scale and software cache coherence on coarse scale could, and is discussed in Section 3.

Changing transparent memory access by introducing hardware incoherent cache imposes at least some redesigning in well known software though, which is generally undesirable. Therefore hardware manufacturers are expected to gain speedups in performance while maintaining the same architecture and hardware cache coherence.

To avoid the introduction of processors with incoherent cache a hierarchical design using multiple stages of inclusive cache, grouping cores into clusters, will be discussed in Section 2. This design is able to work with sufficiently small overhead on up to 512 core-processors as Martin et al. [8] states and does not break with hardware cache coherence.

A cache hierarchy [8] can have a heavy impact on performance, if multi threaded applications are run on distant cores in distant clusters. In order to reduce this issue, various approaches for better thread mapping to hardware cores are discussed in Section 4. This is done without the need of any changes on the applications itself.

2. HARDWARE COHERENCE

Hardware cache coherence in mainstream CPUs is necessary in order to maintain legacy support of software. It may also be a well functioning design in future. As of today mainstream CPUs use inclusive coherent caches[10] with multiple levels. The amount of last level pages usually exceeds the aggregate of lower levels. Intel's Core i7 6700K for instance uses a last level inclusive cache with 8 times of the lower level aggregate.

In this section coherent cache design, it's necessities, advantages and disadvantages will be discussed. As Martin et al. [8] showed in their theoretical paper, there is a possibility that " [..] on-chip coherence is here to stay ". Boyd-Wickizer et al.[3] demonstrated that it is in fact possible to maintain the current OS and application design, even on a 48 Core machine, keeping a reasonable speedup. The following discussion is based on a inclusive, shared design, using exact tracking of sharers in the shared cache (every cache above L1).

Network-traffic is one concern, that needs to be addressed. If blocks are shared and sharers are tracked with one bit per core the traffic per miss is constant (regarding the number of cores) in a scenario of shared blocks. Thus the cost of hardware coherence will be acceptable. If blocks are not shared, messages need to be passed on every miss in a private cache. For clean blocks, those which have not been written,

t raff c cost of cache misses.

To calculate traff c, we must assume values for the size of addresses and cache blocks (such as 8B physical addresses and 64B cache blocks). Request and acknowledgment messages are typically short (such as 8B) because they contain mainly a block address and a message type f eld. A data message is signif cantly larger because it contains both an entire data block plus a block address (such as 64B +8B =72B).

	Clean block	Dirty block
Without coherence	(Req+Data) +0 =80B/miss	(Req+Data) +Data =152B/miss
With coherence	(Req+Data) +(evict+Ack) =96B/miss	(Req+Data) +(Data+Ack)=160B/miss
Per-miss traff c overhead	20%	5%

Figure 1: Example of network traffic cost. With and without coherence [8]

two messages need to be passed: One with the request from a cores private cache to the shared cache and one with the shared cache response containing the data. If a block has been written (a dirty block) in a private cache before an other core hits a read miss on it, then the dirty block needs to be written back to the shared cache and forwarded to the requesting core, that requested the miss. These cases are independent of the number of cores, too and thus negligible for scaling. An example for traffic cost can be seen in the following Figure 1 In this Figure the exact amount of traffic per miss is calculated for 64Byte blocks, with and without coherence.

Since there is fewer traffic with block *sharing* and the exact amount can be tracked, this design choice is preferable. Exact tracking of sharers can lead to an enormous amount of storage overhead tough. With a block size of 64 Bytes and 1024 Cores the overhead amounts to *50%* of total storage cost for tracking of sharers only. This much overhead is not deemed acceptable, therefore a hierarchical model[8] dealing with this problem will be introduced in Section 2.1.

Figure 2 shows the advantages of exact sharer tracking (a) against hybrid tracking (b), having 32 bits for each block representing cores. If the amount does not exceed 32 - or if exceeded, an inexact representation is used, which increases traffic until the core count is high enough that messages need to be passed on every miss to every core anyway.

Therefore a exact representation of sharers is favorable. It's storage costs can be dealt with a hierarchical design[8]. See Subsection 2.1.

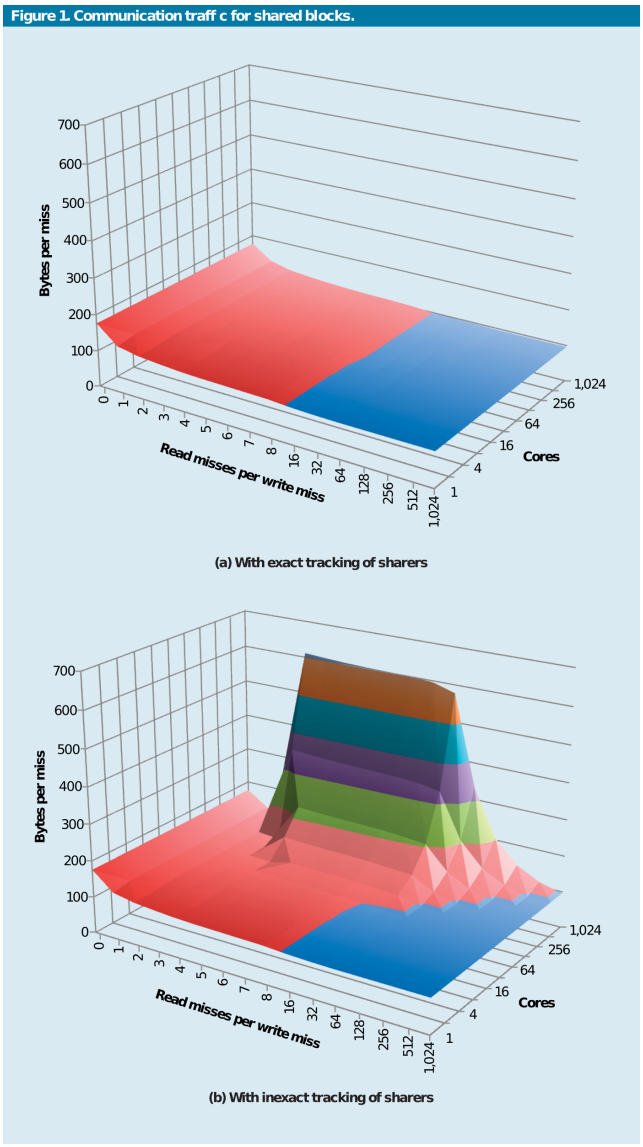

Figure 2: Traffic for shared blocks for write misses against read misses. (a) illustrating exact tracking of sharers, while (b) illustrates non exact tracking with 32 tracking bits. [8]

2.1 Hierarchical Design

A hierarchical design [8] for cache coherence can overcome the extreme overhead for exact tracking of sharers. It exploits the fact, that one hierarchy level only needs to track its underlying sharers per block. Figure 3 (shortened to save space) illustrates the following Example: If you consider 144 cores, a 3 level hierarchy with inclusive caches, a design choice may be 12 cores with private caches, having a shared cache per cluster. Then each level 2 cache only needs 12 bit for exact tracking of block sharers. The last level again needs only 12 bit to encode each cluster. Thus every block can be tracked precisely, on the cost of higher latency if a block is shared across multiple clusters. This issue may be overcome software is aware of the hardware

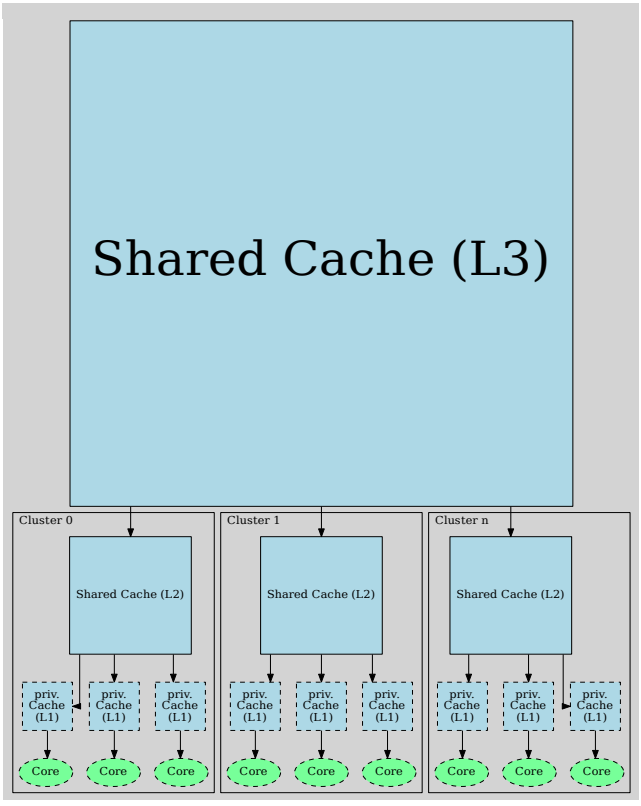

Figure 3: Illustration of a hierarchical cache design. The area of boxes representing caches is roughly proportional to it's actual size.

architecture. An example may be that the operating system schedules processes that need a large amount of communication on one cluster if possible. Thus inter process messages and synchronization is handled faster and does not need escalation to a higher cache level. Further discussed in Section 4.

2.2 Accelerators on heterogeneous Processors

Power et al. [9] describes a model (Heterogeneous System Coherence) for "hardware coherence between CPUs and GPUs", which can be implemented efficiently. This eases the burden for programmability, eliminating the need of explicit copies for the Graphics Processing Unit. Their model is specialized in modern All Processing Units (APU's) that incorporate a classic CPU and GPU. Intel's i-Series [6] already are only APU's and AMD also has a series of APU hybrid processors [2]. Less and more efficient traffic result in less energy usage and thus is favorable for Processors in general but especially for energy saving ones in mobile usage, where APU's are used to save power, since the total amount of available energy is limited.

The GPU and CPU part of modern hybrid processors share the same virtual address space: Both may use the systems ram. Therefore explicit copies are not favorable, since they need more ram and handling of multiple address spaces. In order to reduce storage cost, caused by explicit copies and traffic while increasing total throughput Power et al. designed HSC[9]. Since locality of code and data is different for each parts - GPU's usually have higher spacial and lower temporal locality than CPU's - HSC implements one region directory for both and region buffers for GPU and CPU L2 caches respectively[9]. The region directory consists of a region tag, state bits, one bit for CPU access and one bit for GPU access. In case of a hierarchical CPU design, it's last level cache may have the region buffer. All misses from each processor part, that escalate above L2 are firstly handled by the region buffer. If permission is found, requests are sent directly to system memory. If not, the region directory handles the request and may grand permission. Regions may consist of 16 to 64 blocks each. Because of this granularity size most of L2 misses are routed directly to system memory, keeping a low workload on the directory. Power et al.[9] achieved an increased performance of factor 2 and 94% less bandwidth on the directory on average.

3. OTHER COHERENCE

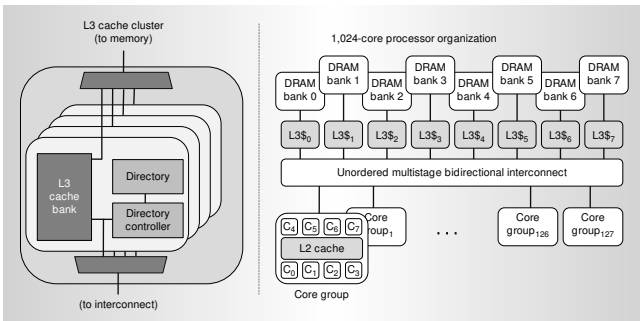

Figure 4: Baseline processor architecture with 1024 cores , 8 per cluster in a RISC-like hierarchical organization [7]

While pure hardware cache coherence is nice to have from the programmers view, there are issues which should be discussed. The issue of hardware cache coherence is, that messages are rather pulled on demand: A miss in a cores private cache is necessary in order to fetch the needed data into it's cache. Therefore the core has to wait until the needed data is available. Software coherence protocols are the opposite and usually work in a push manner: They save blocks in the according private caches whenever writes occur, which makes short delays possible but possibly on the expense of traffic, since the pushed data may be unread by some cores.

Kelm et al. proposed a hybrid design which they called *Cohesion* [7]. This system utilizes hardware based coherence for fine granularity at block level and software coherence for coarse granularity in order to tackle the problem of traffic and storage overhead on many core processors.

In their simulation Software cache coherence needed significantly less messages to be passed between locally shared L2 cache and global L3 for most of the tested algorithms. Figure 8 illustrates the results, normalized to software coherence. False sharing, which is the *false* sharing of data in a block that is actually not shared, can be prevented by fine grained software coherence protocols. An example for false sharing are two variables stored close together in address space that are stored in a shared block and used by a concurrently running program. The first of these variables is used by one thread only and the second by the other but the block containing both of them is shared nevertheless. Software Cache Coherence can eliminate this issue, since it is knowledgeable of the actual information shared.

Hardware cache coherence has the advantage of ensuring latest data for every core in case of shared blocks. This gives the programmer memory transparency and simplifies the actual work to be done. Programs can be ported to other hardware cache coherent systems without much effort and if at all only minor rewrites compared to a totally different system. The total performance may be impacted though, since processors might have varying implementa-

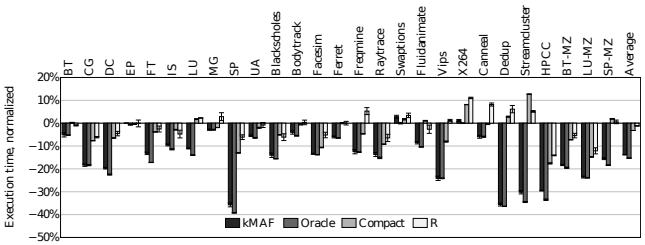

Figure 5: Execution time normalized to OS.[5]

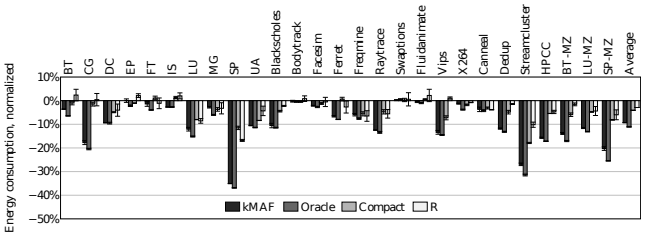

Figure 6: Energy consumption normalized to OS.[5]

tions for hardware coherence, although they appear to have the same architecture from the software developers view. An example are the difference between AMD and Intel CPU's, although they both have the "same" architecture.

Since neither hardware nor software cache coherence are perfect, the by Kelm et al. proposed design could be a perfect alternative utilizing the benefits of both. By not enforcing hardware cache coherence only, the transition between an general purpose processor and specific accelerators like GPUs can also be handled by cohesion, unifying access to different processors. This may actually improve programmability.

Figure 4 shows the processor being used for the simulations by Kelm et al. [7]. It is organized in 128 clusters of 8 cores each. Every core has it's own L1 cache, sharing a unified L2 cache within a cluster. The L3 cache banks are non inclusive to the L2 ones. A directory is used that holds entries cached in at least one L2. This means that the directory may contain lines which are not cached in L3. For software coherence a region table, build on chip, is employed that holds address ranges for software coherence.

Since hardware coherence and software coherence need to be organized, cohesion acts as a bridge between these two protocols. Figure 7 illustrates the transitions from hardware to software coherence. Cohesion acts as link between both implementations for coherence by moving blocks, kept coherent by software to hardware managed coherence. This can be achieved by only one extra *incoherence*-bit in each L2 cache on the proposed processor. If a block is stored *coherently*, queries can be handled by hardware coherence. On every request from L2 to L3 Cohesion queries the directory. On directory and L3 hit the L2's request can be handled with hardware coherence. On directory hit but L3 miss access is blocked by the directory and the region tables are used. On fill from memory the directory sends a response that the requested data is accessible. If a request has neither a directory, nor a L3 hit, L3 sends a response to L2 that an incoherent access has occurred. L2 sets it's incoherence bit accordingly on arrival of the response. Then cohesion can handle the request with it's software side.

4. PROGRAMMING

The Programmability of Hardware is and will still be an important factor of it's success. Even the most advanced and performant hardware will probably not be used if it's complexity in programming is unacceptable. Therefore the pressure not to break with established pragmatisms is high and backward compatibility deemed necessary. As described in subsection 2.1 it is indeed possible not to break with existing designs and stay cache coherent, even when scaling to many core systems. Thus a existing software ecosystem can still be used while further speedups are achievable with minor modifications. Two possible software modifications are presented in the following paragraphs.

Cruz et al.[4] proposed a system for "Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols"[4]. They gained *13.9%* in execution time while having *30.5%* less cache misses and *39.4%* less invalidation messages. The speedups and less traffic as result of less cache misses are made possible by "[..] communication aware thread mapping". Cruz exploited modern CPU's cache hierarchy as implicated in Section 2.1. A multilevel hierarchical cache like illustrated in 3 and higher order hierarchies may benefit even more from Cruz's thread mapping than the tested ones, since latencies for synchronization between distant cores increase with hierarchy level. Hardware awareness may become inevitable for high throughput on many core systems.

Similar results were achieved by Diener et. al[5], who gained an average execution time reduction of *13.8%* and improved energy efficiency by *9.3%* on average. These results were achieved by "[..]Kernel-Level Management of Thread and Data Affinity" (kMAF) [5] and are therefore favorable, since no program modifications are necessary. This was done by analyzing memory access characteristics of parallel applications and optimizing locality of threads on hardware (thread affinity) with this information. No previous information of the applications behavior is necessary, making the usage of kMAF easy for existing software. Figure 5 and Figure 6 show their measurements. Although these results were generated for a small set of benchmarks and were not implemented system wide, they appear to be very promising.

Cohesion, an alternative approach for cache on many core systems, as described in Section 3 may have huge advantages in scaling but well known ease of programmability for general purpose processors may suffer. On the other hand Kelm et al. [7] suggested the implementation of Cohesion by compiler, runtime or programming models. Thus they "[..] make coherence management and optimization a choice rather than a burden for correctness" [7] and give the programmer an easy entry while having the tools for optimizations, if wanted.

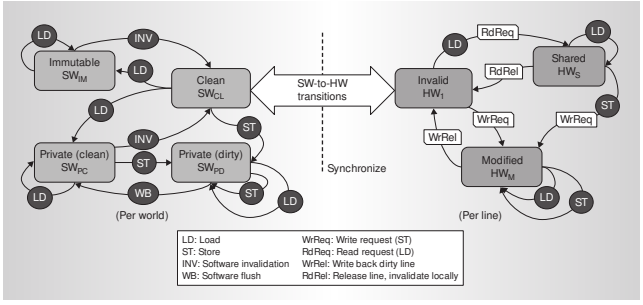

Figure 7: Cohesion as bridge between hardware and software coherence. Left: Software coherence protocol; Right: Hardware coherence protocol.[7]

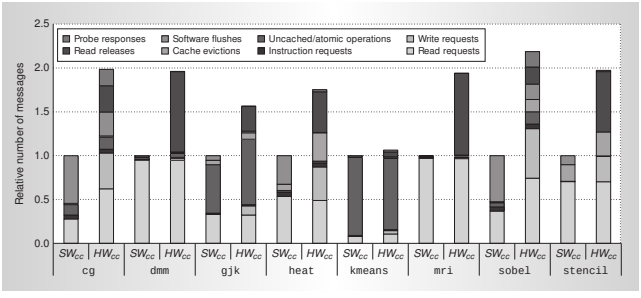

Figure 8: Number of messages sent from L2 to L3 cache (for various algorithms) on a machine shown in 4. Normalized to Software Cache Coherence protocol.[7]

5. CONCLUSION

Two different models, a hierarchy for hardware coherent caches[8] and a hybrid cache coherence design[7] were presented here. Whilst both have advantages, they also have issues that need to be dealt with.

Hardware coherence as proposes by Martin et al.[8] has the enormous benefit of not breaking with current hardware coherence models and thus giving the possibility to use current and legacy software without any modifications. Thus the already proven ecosystem of operating systems and software can simply be used further on. There might be impacts on performance though if software is running concurrently on distant clusters, causing a lot of high latency traffic when synchronizing. Cruz et al.[4] and Diener et al.[5] proposed systems to overcome those problems on software side though.

Kelm et al.[7] proposed a system they called *Cohesion*. It combines both: *Hardware* and *software* cache coherence giving. With this technique even many core processors can achieve high throughput. Their proposed processor is organized in clusters implementing hardware coherence. The clusters are kept coherent by software. While it's clear advantage is a low amount of overhead, it also imposes a burden on the programmer, since software *can* now manage hardware's caches and probably need to if good performance is necessary: An application programmer needs to focus more on the hardware and not only on the application's features itself, when using cohesion. Then the efficiency is very high though and false sharing can be eliminated on coarse grain level. Some restructuring of software will also be necessary.

Both systems are a legitimate design choice and may be implemented in future. Since both studies were only in theory or simulation, a real implementation of them might show side effects not covered. Since the trend in mainstream processors was some kind of hardware coherent implementation it will unlikely be abandoned. Therefore the hierarchical design by Martin et al. is probably favored.

6. REFERENCES

[1] Whitepaper nvidia geforce gtx 1080.

[2] Advanced-Micro-Devices:(AMD). Amd apu public website, 2016.

[3] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An analysis of linux scalability to many cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[4] E. H. Cruz, M. Diener, M. A. Alves, and P. O. Navaux. Dynamic thread mapping of shared memory applications by exploiting cache coherence protocols. *Journal of Parallel and Distributed Computing*, 74(3):2215–2228, 2014.

[5] M. Diener, E. H. Cruz, P. O. Navaux, A. Busse, and H.-U. Heiß. kmaf: Automatic kernel-level management of thread and data affinity. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, PACT '14, pages 277–288, New York, NY, USA, 2014. ACM.

[6] Intel. Intel ark i7 6th generation, 2016.

[7] J. H. Kelm, D. R. Johnson, W. Tuohy, S. S. Lumetta, and S. J. Patel. Cohesion: An adaptive hybrid memory model for accelerators. *IEEE Micro*, 31(1):42–55, Jan. 2011.

[8] M. M. K. Martin, M. D. Hill, and D. J. Sorin. Why on-chip cache coherence is here to stay. *Commun. ACM*, 55(7):78–89, July 2012.

[9] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 457–467, New York, NY, USA, 2013. ACM.

[10] I. Singh, A. Shriraman, W. W. L. Fung, M. O'Connor, and T. M. Aamodt. Cache coherence for gpu architectures. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 578–590, Feb 2013.