Cache Line Aware Optimizations for ccNUMA Systems

Sabela Ramos
Computer Architecture Group
University of A Coruña
Spain
sramos@udc.es

Torsten Hoefler
Scalable Parallel Computing Lab
ETH Zurich
Switzerland
htor@inf.ethz.ch

ABSTRACT

Current shared memory systems utilize complex memory hierarchies to maintain scalability when increasing the number of processing units. Although hardware designers aim to hide this complexity from the programmer, ignoring the detailed architectural characteristics can harm performance significantly. We propose to expose the block-based design of caches in parallel computers to middleware designers to allow semi-automatic performance tuning with the systematic translation from algorithms to an analytic performance model. For this, we design a simple interface for cache line aware (CLa) optimization, a translation methodology, and a full performance model for cache line transfers in ccNUMA systems. Algorithms developed using CLa design perform up to 14x better than vendor and open-source libraries, and 2x better than existing ccNUMA optimizations.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques

General Terms

Design, Algorithms, Performance

Keywords

Cache coherence, multi-cores, performance modeling.

1. MOTIVATION

Today's multi- and many-core architectures provide the illusion of coherent shared memory to simplify the initial design of parallel programs from a serial specification. Cache coherence protocols guarantee that there is exactly one value in each memory location in the system, even when several threads write simultaneously. To achieve the highest performance, programmers need to design highly-scalable parallel algorithms to utilize the exponentially growing number of cores. While cache coherence simplifies the initial design, the complexity of the protocol's performance characteristics often leads to poorly-scalable solutions. This is mainly because the complex interactions in the cache coherence protocol are hidden from programmers, essentially forming an *abstraction barrier for performance-centric programming.* To overcome this, we propose the use of a Cache Line aware (CLa) design, a simple abstraction that considers cache coherence hardware during algorithm-design and implementation. The main features of CLa's abstract machine model are detailed cost functions for accesses in coherent hierarchical non-uniform memory access (ccNUMA) computers.

In summary, the specific contributions of our work are: (1) we propose Cache Line aware (CLa) optimization, a method for performance-centric programming of ccNUMA systems; (2) we show how to systematically model the performance of algorithms analytically; (3) we design a methodology to translate shared memory communication algorithms directly into an analytical performance model; and (4) we conduct a practical study with a dual-socket architecture yielding speedups between 1.8x and 14x over optimized libraries.

We start with a brief discussion of a performance model for cache-coherence protocols in ccNUMA systems. In Section 3 we introduce the rules for translating an algorithm into the performance model. Section 4 describes and exemplifies the application of our principles for CLa design.

2. CLA PERFORMANCE MODELS

Modern multi- and many-core processors provide cache coherence as a means to hide the management of data-sharing among cores. Yet, we show that performance engineers need to reason explicitly about cache coherence transfers in order to obtain highest performance. We propose a performance model based on a set of building blocks that enables programmers to analyze algorithms in terms of (cache) line transfers. We identify two main primitives which we parametrize through benchmarking considering thread location and coherence state: single-line and multi-line transfers. However, various interactions between threads may introduce additional overheads. Some interactions, such as contention (several threads accessing the same cache lines) and congestion (several threads accessing different lines) can be benchmarked and modeled accurately. Other interactions depend on the real-time order in which operations are performed and are not predictable (see Section 2.3).

Although our conclusions and methods are not limited to a specific architecture, we now briefly describe the NUMA system on which we developed and executed our benchmarks, a dual-socket eight-core processor Intel Sandy Bridge Xeon

HPDC'15, June 15–20, 2015, Portland, Oregon, USA.

ACM 978-1-4503-3550-8/15/06 ...$15.00.
http://dx.doi.org/10.1145/2749246.2749256.

E5-2660, at 2.20GHz with Hyper Threading activated and Quick Path Interconnect (QPI, 8 GT/s). Each socket has three levels of cache. L1 (32 KB data and 32 KB instructions) and L2 (256 KB unified) caches are private to each core. An L3 cache (or LLC) of 20 MB is shared by all cores within a socket and divided in eight slices. The internal components of the chip, including the LLC slices, are connected via a ring bus. All cores can use every LLC's cache slices, thus having access to data stored in any of them. The chip uses the MESIF cache coherence protocol [13], based on the classic MESI (Modified-Exclusive-Shared-Invalid) protocol. It adds a *Forward* state to optimize the management of shared states. Although it globally behaves like a snooping protocol, cache coherence within each socket is kept by the LLC, that holds a set of bits indicating the state of the line and which cores have a copy. Among sockets, the QPI implements the cache coherence protocol. It is in this scenario when the F state avoids multiple answers from different sockets to a snooping request.

2.1 Single-line Transfers

The basic block in our model is the transfer of a cache line between two cores. Line transfers are caused by two operations: *read* and *RFO* (Read For Ownership). Both involve fetching lines, but the latter indicates the intention to write. We estimate the cost of both as a *read* (R) although there could be some differences, e.g., an *RFO* of a shared line means that all copies must be invalidated. But we first analyze transfers between two threads where this difference is not significant.

We implemented a pingpong data exchange to analyze the impact of thread location and coherence state. Results show that there are significant differences when varying the location of threads and lines. But there are only minor variations for different cached states, hence, we cluster the costs for line transfers in five classes: (1) L if the line is in local cache, (2) R if it is in another core from the same socket, (3) Q if it is in another socket, (4) I if it is in memory in a local NUMA region, and (5) QI if it is in a remote NUMA region. Such a model would need to be generated for each microarchitecture. We parametrize the cost of a line transfer for each class with BenchIT [13], obtaining the following latencies: (1) $R_L \simeq 2.3ns$, (2) $R_R \simeq 35ns$, (3)$R_Q \simeq 94ns$, (4) $R_I \simeq 70ns$, and (5) $R_{QI} \simeq 107ns$.

Sandy Bridge supports loading half lines [1, §2.2.5.1] which is cheaper than always loading full lines. However, other architectures always transfer entire lines, thus, we will work with full lines for generality and clarity.

We use two benchmarks to evaluate contention (threads accessing the same CL) and congestion (threads accessing different lines). In these benchmarks, threads read an external send buffer and copy it into a local receive buffer. We did not observe contention in any scenario and results show no congestion for intra-socket transfers. We further analyze QPI congestion together with multi-line cache transfers.

2.2 Multi-line Transfers

We evaluate multi-line transfers with two benchmark strategies: pingpong and one-directional transfers (similar to those used for contention and congestion). Pingpong times exhibit significant variability when using invalid lines, especially for QPI transfers. This variability stems from the use of DRAM and different NUMA regions and we developed approximate models to simplify algorithm optimization and comparison. Without loss of generality, we work with cached multi-line transfers for which we empirically parametrize the model in Equation (1). N is the number of lines, n is the number of simultaneously accessing threads, and q, o, c are architecture-specific parameters (cf. Table 1). The term cnN represents congestion in the QPI link (it is zero in intra-socket scenarios).

$$T_m(n, N) = q + oN + cnN \tag{1}$$

Table 1: Parameters for multi-line transfer of cached lines

	q $[ns]$	**o** $[\frac{ns}{line}]$	**c** $[\frac{ns}{line \cdot thread}]$	$\mathbf{R}^2$
Intra socket	63.4	11.1	0	0.8
Inter socket	180.65	7.5	3.0	0.91

2.3 Invalidation and Cache Line Stealing

The described building blocks can be used to model algorithms in terms of line transfers but we need to consider two additional sources of overhead due to interactions between threads or cores. First, an *RFO* of a shared line involves invalidation at its n owners (on our test system, it costs nR_R instead of R_R). Second, *cache line stealing* appears when several threads write one line where another thread is polling (*n-writers*), or when several threads poll a line that another thread writes (*n-readers*). Both scenarios get more complex with more than one reader or writer, respectively. To capture all these variations, we use *min-max* models [15] that provide performance bounds by estimating the best and worst stealing scenario. We represent a CLa algorithm in terms of line transfers and thread interactions, using the parametrized building blocks to derive models for the minimum (T_{min}) and maximum (T_{max}) scenarios.

3. A CANDIDATE CLA INTERFACE

In order to expose cache coherence interactions and apply our performance model, we propose a simple methodology that starts by expressing algorithms in a cache line centric manner using primitives that can be implemented in various ways in most languages. We implement them with direct load/store ISA instructions. When they are used for synchronization, we use atomic instructions for writing, but not for reading and polling, because atomics often force the eviction of lines from other caches. The cost of each operation is expressed in terms of location and state of the given lines. For more than one line, we use Equation (1). We define the following operations:

1. `cl_copy(cl_t* src,cl_t* dest,int N)` copies N lines from `src` to `dest`.
2. `cl_wait(cl_t* line,clv_t val,op_t comp=eq)` polls until `comp(*(line),val)` is `true`.
3. `cl_write(cl_t* line,clv_t val)` writes `val` in `line`.
4. `cl_add(cl_t* line,clv_t val)` adds `val` to `line`.

Once we have the CLa pseudo-code, we construct a graph in which nodes are the CLa operations performed by each thread, linked by four types of edges:

E1 A sequence of operations within a thread, represented by dotted directed edges.

E2 Logical dependencies between threads (i.e., reading or polling a line that has been written by others), represented by directed edges.

- E3 Sequential restrictions between threads without order, represented by non-directed edges. We use them when several threads write the same flag (in any order), and a thread polls this flag waiting for all writes.
- E4 Line stealing in non-related operations (e.g., a `wait` on a line that is written in two different stages of an algorithm), represented by dotted lines.

Next, we assign costs to the nodes using the following rules:

1. Flags are initially in memory. First fetch costs R_I.
2. An access to data in local cache costs R_L.
3. The access to the same line by the same thread in consecutive operations is counted once.
4. If the operation has an incoming edge from another thread, it costs R_R or R_Q depending on the location of threads.
5. Read operations with incoming edges from the same node can be simultaneous without contention. E.g., threads copying a line written by another.

In order to derive the T_{min} (cost of the critical path), we define a path as a sequence of nodes linked by E1, E2, and E3 edges, starting in a node with no incoming E1 and E2 edges, and finishing in a node with no outgoing edges. Regarding E3 edges, they link all sequential writes that have outgoing E2s towards the same `wait`. When searching for the critical path, we analyze reorderings of these writes, ensuring that the path visits once each of them before going towards the `wait`. When some E3s represent inter-socket communications, we select the reordering with less QPI transfers.

To identify QPI congestion, we look for directed arrows between sockets that have: (1) different start and end points (accesses to different addresses by different threads), and (2) previous paths of similar cost (simultaneous transfers). Finally, T_{max} is calculated by analyzing line stealing (the main cause is the `wait` operation) and we can refine it by considering which operations might not overlap. We optimize for T_{min} because T_{max} is usually too pessimistic. If an algorithm receives a communication structure as parameter, we analyze multiple structures to obtain the best one.

This set of rules is enough to derive graphs and performance models for multiple communication algorithms and it is easily extensible to cover other interactions.

4. SINGLE-LINE BROADCAST

Broadcast consists of transferring data from one thread (*root*) to n others. We designed a tree-based algorithm taking into account that all children of a given node copy the data without contention. However, the more children a parent has, the more costly the synchronization is: The parent notifies that data is ready (one-to-many, T_{o2m}) and children notify that they have copied so the parent can free the structure (many-to-one, T_{m2o}). We use notification with payload for the one-to-many synchronization. Regarding the many-to-one, we use one line in which children write and the parent polls. Although other analyzed variants have the same T_{min}, this version has lower T_{max}. Our algorithm (cf. Figure 1) uses a generic tree in which each node i has k_i children that copy the same data. We generate all structurally equivalent rooted trees [10], calculating the broadcast latency to select the best structure. This tree could change slightly in a scenario with contention in which we may have rounds of children accessing the same data at different stages.

```
     Function OneLineBroadcast(int me, cl_t * mydata, tree_t tree)
         if tree.parent != -1 then
[S1]         cl_wait(tree.pflag[tree.parent],1);          //one-to-many
[S2]         cl_copy(tree.data[tree.parent],mydata,1);
         if tree.nsons > 0 then
[S3]         cl_copy(mydata,tree.data[me],1);
[S4]         cl_write(tree.pflag[me],1);                  //one-to-many
[S5]         cl_wait(tree.sflag[me],tree.nsons);          //many-to-one
         if tree.parent != -1 then
[S6]         cl_add(tree.sflag[tree.parent],1);           //many-to-one
         end
     end
```

Figure 1: One line broadcast in CLa pseudo-code. The first `if` block corresponds to children polling their parent's flag and copying the data. In the second one, a parent sets the data and the flag, and waits for its children to copy. Finally, children notify to their parent that they have copied.

For a given tree structure, we construct the CLa graph and search the critical path. Figure 2 shows an example of a four-node binary tree (the critical path has thicker edges and nodes with dotted circles). The E1s link operations within each thread and we use E2s in the synchronizations and data copies. Finally, there is an E3 because t_1 and t_2 write the same flag, where t_0 polls.

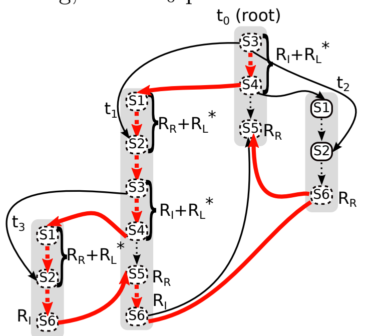

Figure 2: CLa graph for a one line broadcast using a four-node binary tree. Costs with '*' represent situations in which the same thread operates consecutively with the same line and the cost of accessing is counted only once.

Since we use tree structures, we observe regularities in the critical path: It includes the transfer of data from the root to its children plus the synchronizations ($T_{lev}(k_0) = T_{o2m}(k_0) + T_{data} + T_{m2o}(k_0)$), plus the cost of the most expensive subtree ($T_{bc}(stree_i)$, the left one in Figure 2). We generalize it in Equation (2)[1]. The minimization balances the number of simultaneous readers, and the notification cost. In a multi-socket broadcast some edges become QPI links. We generate permutations of the tree nodes to locate the QPI links in different edges[2] and we apply Equation (2) considering: (1) Inter-socket transfers cost R_Q. (2) To isolate QPI transfers and minimize line stealing, we use one synchronization line per socket. And (3) we do not consider QPI congestion caused by different subtrees because our experiments showed that the benefits are minimal.

$$\begin{aligned} &\underset{k_i}{\text{minimize}}\ \ T_{bc}(tree) = T_{lev}(k_0) + \max_{i=1,\dots,k_0}\left(T_{bc}(stree_i)\right) \\ &\text{subject to}\quad T_{bc}(leaf) = 0 \quad \textstyle\sum_{i=0}^{n} k_i = n,\ k_i \geq 0 \end{aligned} \tag{2}$$

Figure 3 shows the performance of our algorithm compared to two MPI libraries and the HMPI NUMA-aware

[1] If we use a global flag where the root sets the shared structure as occupied by this operation, we add R_I

[2] We do not need all permutations: there is no difference among threads from the same socket.

broadcast [12] (using a thread-based implementation to compare algorithms directly). We use RDTSC intervals [14] to synchronize threads before each iteration and we force synchronization data-structures and user-data in the desired cache states. The system used is described in Section 2, with CentOS 6.4. Compilers are Intel v.13.1.1 and GNU v.4.4.7, and MPI libraries Intel MPI v.4.1.4 and Open MPI 1.7.2. The shaded area represents the min-max model. The results of our algorithm include boxplots to represent the statistical variation of the measurements. We schedule up to eight threads in one socket and the rest of them in the second one. Broadcasts with an imbalanced number of threads per socket (e.g., ten threads) use different trees depending on the root is location. Our algorithm clearly outperforms both MPI libraries obtaining a speedup of up to 14x. HMPI uses a flat tree with synchronization based on barriers. We improve this approach by up to 1.8x.

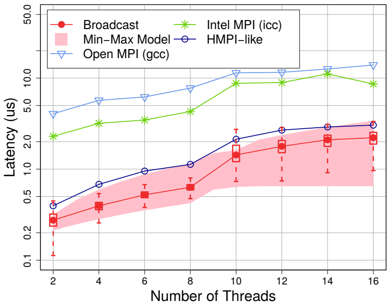

Figure 3: Single-line broadcast

5. RELATED WORK

Analytical performance models have been largely used to optimize parallel computation [3, 7], especially in distributed environments. Some of them were extended with memory concerns [5, 6] and multi-core features [11, 16]. Model-driven algorithm optimization has been tackled in multiple works [9, 12] but with almost no cache coherence concerns. And most cache coherence works focus on memory hierarchy and cache conflicts [2, 4]. David et al. [8] make an exhaustive analysis of lock synchronization for different multi-core architectures, evaluating the effect of cache states and memory operations. Our performance model significantly improves our previous work [15] on homogeneous many-core processors. We extend the model for hierarchical NUMA machines, generalizing its applicability and the algorithm design.

6. CONCLUSIONS

While cache coherence simplifies the use of multiple cores, it exhibits complex performance properties and thus complicates high-performance code design. We address this issue with cache line aware (CLa) optimizations, a semi-automatic design and optimization method that eases the translation of an algorithm to a performance model in a systematic manner. We exemplify its use improving broadcast performance up to 14x in comparison to highly-optimized vendor-provided communication libraries, and up to 1.8x when compared to the NUMA-aware HMPI collectives. Moreover, we expect higher improvements in future many-core systems. We expect that model-driven algorithm engineering will benefit high-performance low-level software engineering, and it will be necessary to address software and hardware complexity in the long run.

7. ACKNOWLEDGEMENTS

We thank the support from Alexander Supalov and Robert Blankenship from Intel. This work was supported by the Ministry of Economy and Competitiveness of Spain and FEDER funds of the EU (Project TIN2013-42148-P).

8. REFERENCES

[1] Intel® 64 and IA-32 Architectures Optimization Ref. Manual, 2014.

[2] A. Agarwal et al. An Analytical Cache Model. *ACM Trans. on Computer Systems*, 7(2):184–215, 1989.

[3] A. Alexandrov et al. LogGP: Incorporating Long Messages into the LogP Model - One Step Closer towards a Realistic Model for Parallel Computation. In *Proc. 7th ACM SPAA'95*, pages 95–105, S. Barbara, CA, USA, 1995.

[4] D. Andrade et al. Accurate Prediction of the Behavior of Multithreaded Applications in Shared Caches. *Parallel Computing*, 39(1):36 – 57, 2013.

[5] K. W. Cameron et al. lognP and log3P: Accurate Analytical Models of Point-to-Point Communication in Distributed Systems. *IEEE Trans. on Computers*, 53(3):314–327, 2007.

[6] K. W. Cameron and X. H. Sun. Quantifying Locality Effect in Data Access Delay: Memory logP. In *Proc. 17th IEEE IPDPS'03*, (8 pages),Nice, France, 2003.

[7] D. Culler et al. LogP: towards a Realistic Model of Parallel Computation. *SIGPLAN Not.*, 28(7):1–12, 1993.

[8] T. David et al. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *Proc. 24th ACM Symp. SOSP'13*, pages 33–48, Farminton, PA, USA, 2013.

[9] R. M. Karp et al. Optimal Broadcast and Summation in the LogP Model. In *Proc. 5th ACM SPAA'93*, pages 142–153, Velen, Germany, 1993.

[10] G. Li and F. Ruskey. Advantages of Forward Thinking in Generating Rooted and Free Trees. In *Proc. 10th ACM-SIAM SODA'99*, pages 939–940, Baltimore, MD,USA, 1999.

[11] L. Li et al. mPlogP: A Parallel Computation Model for Heterogeneous Multi-core Computer. In *Proc. 10th IEEE/ACM Intl. CCGRID'10*, pages 679–684, Melbourne, Australia, 2010.

[12] S. Li et al. NUMA-aware Shared-memory Collective Communication for MPI. In *In Proc. 22nd Intl. Symp. HPDC'13*, pages 85–96, New York, NY, USA, 2013.

[13] D. Molka et al. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *Proc. 18th Intl. Conf. PACT'09*, pages 261–270, Raleigh, NC, USA, 2009.

[14] S. Ramos and T. Hoefler. Benchmark Suite for Modeling Intel Xeon Phi. http://gac.des.udc.es/~sramos/xeon_phi_bench/xeon_phi_bench.html.

[15] S. Ramos and T. Hoefler. Modeling communication in cache-coherent SMP systems: a case-study with Xeon Phi. In *Proc. of the 22nd Intl. HPDC'13*, pages 97–108, New York, New York, USA, 2013.

[16] Z. Wang and M. F. O'Boyle. Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. In *Proc. 14th ACM SIGPLAN Symp. PPoPP'09*, pages 75–84, Raleigh, NC, USA, 2009.