Evaluating the Cost of Atomic Operations on Modern Architectures

Hermann Schweizer
Dept. of Computer Science
ETH Zurich
hermannschweizer@hotmail.com

Maciej Besta
Dept. of Computer Science
ETH Zurich
maciej.besta@inf.ethz.ch

Torsten Hoefler
Dept. of Computer Science
ETH Zurich
htor@inf.ethz.ch

Abstract

Atomic operations (atomics) such as Compare-and-Swap (CAS) or Fetch-and-Add (FAA) are ubiquitous in parallel programming. Yet, performance tradeoffs between these operations and various characteristics of such systems, such as the structure of caches, are unclear and have not been thoroughly analyzed. In this paper we establish an evaluation methodology, develop a performance model, and present a set of detailed benchmarks for latency and bandwidth of different atomics. We consider various state-of-the-art x86 architectures: Intel Haswell, Xeon Phi, Ivy Bridge, and AMD Bulldozer. The results unveil surprising performance relationships between the considered atomics and architectural properties such as the coherence state of the accessed cache lines. One key finding is that all the tested atomics have comparable latency and bandwidth even if they are characterized by different consensus numbers. Another insight is that the hardware implementation of atomics prevents any instruction-level parallelism even if there are no dependencies between the issued operations. Finally, we discuss solutions to the discovered performance issues in the analyzed architectures. Our analysis enables simpler and more effective parallel programming and accelerates data processing on various architectures deployed in both off-the-shelf machines and large compute systems.

1. Introduction

Multi- and manycore architectures are established in both commodity off-the-shelf desktop and server computers, as well as large-scale datacenters and supercomputers. Example designs include Intel Xeon Phi with 61 cores on a chip installed in Tianhe-2 [17], or AMD Bulldozer with 32 cores per node deployed in Cray XE6 [29]. Moreover, the number of cores on a chip is growing steadily and CPUs with hundreds of cores are predicted to be manufactured in the foreseeable future [5]. The common feature of all these architectures is the increasing complexity of the memory subsystems characterized by multiple cache levels with different inclusion policies, various cache coherence protocols, and different on-chip network topologies connecting the cores and the caches [8].

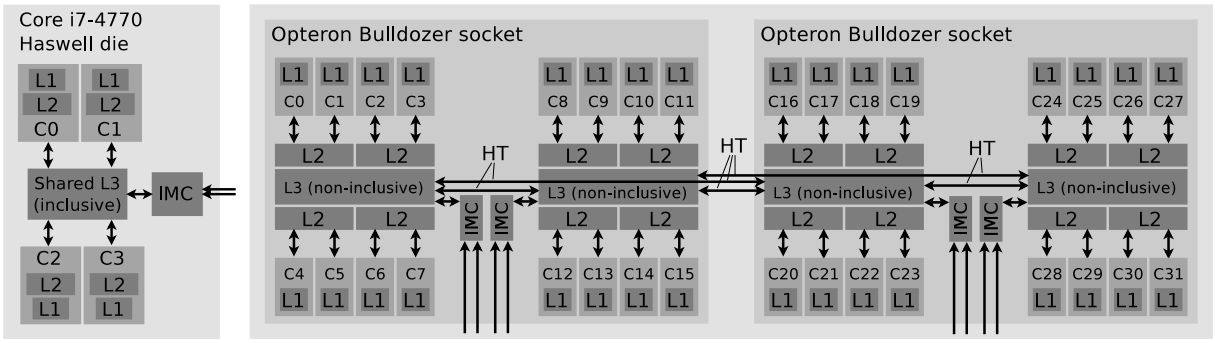
Virtually all such architectures provide atomic operations that have numerous applications in parallel codes. Many of them (e.g., Test-and-Set) can be used to implement locks and other synchronization mechanisms [10]. Others, e.g., Fetch-and-Add and Compare-and-Swap, enable constructing miscellaneous lock-free and wait-free algorithms and data structures that have stronger progress guarantees than lock-based codes [10].

Despite their importance and widespread utilization, the performance of atomic operations has not been thoroughly analyzed so far. For example, according to the common view, Compare-and-Swap is slower than Fetch-and-Add [22]. However, it was only shown that the semantics of Compare-and-Swap introduce the notion of "wasted work" resulting in lower performance of some codes [9, 22]. Other works provide basic insights and illustrate that the performance of atomics is similar on multi-socket systems due to the overheads from socket-to-socket hops [4]. Yet, to the best of our knowledge, no model and benchmarks analyze in detail the latency or bandwidth of the execution of the actual operations in the context of complex multilevel cache and memory hierarchies. Even more importantly, the performance tradeoffs between atomics and various characteristics of multi- and manycore systems (cache coherency protocol, number of memory hierarchy levels, etc.) have also not been thoroughly studied so far. For example, a single node in popular Cray XE6 cabinets provides two AMD Bulldozer CPUs connected with a HyperTransport (HT) link, each CPU consists of two dies also connected with HT, and each die provides one L3 cache and four L2 caches shared by eight cores [29]. It is unclear what the performance of different atomics is on such a system, what is the influence of the cache coherency protocol, what is the performance impact of mechanisms such as adjacent cache line prefetchers, and whether optimizations such as instruction-level parallelism are available for atomics.

In this paper, we introduce a performance model and establish a methodology for benchmarking atomics. Then, we use it to analyze the latency and bandwidth of the most popular atomic operations (Compare-and-Swap, Fetch-and-Add, Swap). Our results unveil undocumented architectural properties of the tested systems and identify several performance issues of the evaluated operations. We discuss solutions to these problems and we illustrate how our model and analysis simplify parallel programming in areas such as graph analytics. The key contributions of this work are:

- We introduce a performance model for the latency and bandwidth of atomics. The model takes into account different cache coherency states and the structure of the caching hierarchy.
- We establish a methodology for benchmarking atomic operations targeting state-of-the-art multi- and manycore architectures with deep memory hierarchies.
- We conduct a detailed performance analysis of Compare-and-Swap, Fetch-and-Add, and Swap. We use the analysis to validate the model, to illustrate undocumented architectural properties of

[Copyright notice will appear here once 'preprint' option is removed.]



(a) The evaluated Haswell.

(b) The evaluated Bulldozer.

Figure 1: (§ 2.2) The illustration of two of the analyzed architectures: Intel Haswell and AMD Bulldozer. We exclude the Intel Ivy Bridge and Xeon Phi testbeds due to space constraints. The former combines the features of the Haswell and Bulldozer testbeds as it hosts private L1s and L2s and offers 24 cores grouped in two CPUs. The latter is a manycore design where each core has a private L1 and L2; the cores form a ring.

the tested systems, and to suggest several improvements in the hardware implementation of respective atomics.

- We discuss how our analysis simplifies parallel programming using the example of graph traversal algorithms.

2. Background

We now present a general approach for benchmarking memory accesses (Sec. 2.1) that we will later use and extend. Then, we discuss the evaluated architectures and atomics (Sections 2.2 and 2.3).

2.1 Benchmarking Memory Accesses

In our analysis we use and extend the X86membench infrastructure for benchmarking memory accesses [8] that utilizes the high resolution `RDTSC` time stamp counter. Each benchmark consists of the following phases:

Preparation: A buffer of the selected size is allocated and filled with the data specific to each benchmark (see more details in Section 3). The TLB is warmed up and the data is placed in caches in the selected coherency state.

Synchronization: This phase makes sure that all threads finished the preparation phase and it defines a future moment in time when all the threads will start the measurement phase.

Measurement: All participating threads: take a time stamp `t_start`, do a measurement, and take the other time stamp `t_end`.

Result collection: The time stamps of all participating cores are communicated and the total time of execution is calculated as `max(t_end) - min(t_start)`.

2.2 Evaluated Architectures and Systems

Next, we present the targeted architectures and systems. We illustrate more details in Figure 1 and Table 1.

Haswell is an Intel state-of-the-art microarchitecture that offers sophisticated mechanisms such as hardware transactional memory (HTM) [31]. In our benchmarks we use a quadcore Haswell chip included in a commodity off-the-shelf server machine; see Figure 1a. The L1 and L2 caches are private to each core and the L3 inclusive cache is shared by all the cores. We select this configuration to analyze a simple commodity multicore system.

Ivy Bridge is an Intel microarchitecture used in various supercomputers such as Tianhe-2 [17] or NASA Pleiades [30]. Here, we evaluate an Ivy Bridge configuration installed in a cluster Euler from ETH Zurich that contains two 12-core CPUs connected with Quick Path Interconnect (QPI). The L1 and L2 are private to each core and the L3 inclusive cache is shared by all the cores. We use this configuration to analyze the performance characteristics of deep memory hierarchies with three cache levels.

Bulldozer is an AMD microarchitecture designed to improve power efficiency for HPC applications [3]. Here, we evaluate a configuration included in the Cray XE6 Monte Rosa supercomputer [29]; see Figure 1b. A compute node deploys two 16-core AMD Bulldozer Interlagos CPUs. Each CPU hosts two 8-core dies that are connected with HyperTransport (HT) [27]. We selected this system to unveil differences between Intel and AMD systems and to analyze the effects coming from a particularly complex design with three cache levels, multiple CPUs, shared L2 caches, and multiple dies per CPU.

MIC is an Intel state-of-the-art manycore architecture deployed in Xeon Phi processors that targets massively parallel systems. We evaluate a configuration with 61 cores. Each core has a private L1 and L2; there is no L3. The cores are connected with a ring topology. We use MIC to analyze a highly parallel coprocessor installed in supercomputing machines such as Tianhe-2 [17].

The considered systems represent both multicore commodity off-the-shelf machines (Haswell) and high-end manycore HPC systems (MIC, Ivy Bridge, Bulldozer). They all use the same cache line size (64B) and use extensions of the well known MESI [21] cache coherency protocol. Haswell and Ivy Bridge utilize MESIF; it avoids redundant data transfers from other cores or memory by adding the **F**orward state to designate a cache to respond to any requests for the given shared line [8]. AMD Bulldozer deploys the MOESI protocol that prevents write-backs to memory by introducing the **O**wned state which allows a dirty cache line to be shared [8]. Finally, Xeon Phi deploys a protocol based on MESI that extends the Shared state with a directory-based cache coherency protocol GOLS (Globally Owned Locally Shared) to simulate the Owned state to enable sharing of a modified line.

Another difference between the tested systems is the structure of L3; we will later show that it impacts the performance of atomics. Xeon Phi hosts no L3. Ivy Bridge and Haswell deploy the inclusive L3 cache where each cache entry contains a *core valid bit* for each core on the CPU. If this bit is set then the related core *may* have the respective cache line in its L1 or L2, possibly in a dirty state. If none of the core valid bits is set (or if the cache line is not present in L3) then the respective cache line is also not present in L1 and L2. On the contrary, L3 in AMD Bulldozer is neither exclusive nor

	Architecture:	Haswell	Ivy Bridge	Bulldozer	MIC
Processor	Manufacturer	Intel	Intel	AMD	Intel
	CPU model	Core i7-4770	Xeon E5-2697v2	Opteron 6272	Xeon Phi 7120
	Cores/CPU	4	12	16(2x8)	61
	CPUs	1	2	2	1
	Core frequency	3400 MHz	2700 MHz	2100 MHz 1238 MHz	
	Interconnect	-	2x QPI (8.0 GT/s)	4x HT 3.1 (6.4 GT/s)	-
Caches	Cache line size	64B	64B	64B	64B
	L1 cache	32KB per core	32KB per core	16KB per core	32KB per core
	L1 Update policy	write back	write back	write through	write back
	L2 cache	256KB per core	256KB per core	2MB per 2 cores	512KB per core
	L2 Update policy	write back	write back	write back	write back
	L2 incl/excl:	neither	neither	neither	inclusive
	L3 cache	8MB fully shared	30MB fully shared	8MB per 8 cores	-
	L3 Update policy	write back	write back	write back	-
	L3 incl/excl:	inclusive*	inclusive*	non-inclusive	-
	CC protocol	MESIF	MESIF	MOESI	MESI-GOLS
Memory	Main memory	8GB	64GB	32GB	8GB
	memory channels/CPU	1x dual channel	2x dual channel	2x dual channel	8x dual channel
	Huge page size	2MB	2MB	2MB	2MB
Others	Linux kernel used	3.14-1	2.6.32	2.6.32	2.6.38.8
	CAS assembly instruction	`Cmpxchg`	`Cmpxchg`	`Cmpxchg`	`Cmpxchg`
	FAA assembly instruction	`Xadd`	`Xadd`	`Xadd`	`Xadd`
	SWP assembly instruction	`Xchg`	`Xchg`	`Xchg`	`Xchg`

Table 1: The comparison of the tested systems. We denote the cache coherency protocol as `CC protocol`. "*" indicates that the shared inclusive L3 cache in Intel Haswell and Ivy Bridge contains a *core valid bit* for each core on the CPU that indicates whether a respective core may contain a given cache line in its private higher level cache (the bit is set) or whether it certainly does not contain this cache line (the bit is zeroed).

inclusive: the presence of a cache line in L2 does not determine its presence in higher level caches. This will have a detrimental effect on the performance of atomics as we will illustrate in Section 5.

2.3 Evaluated Atomics Operations

Finally, we discuss the selection of the evaluated atomics.

Compare-and-Swap(*mem, reg1, reg2) (CAS): it loads the value stored in *mem into reg1. If the original value in reg1 is equal to *mem then it writes reg2 into *mem. We select CAS because it is utilized in numerous lock-free and wait-free data structures and algorithms [10] as well as various graph processing codes such Graph500 [23].

Fetch-and-Add(*mem, reg) (FAA): it fetches the value from a memory location *mem into a register reg and adds the previous value from reg to *mem. We selected FAA because of its importance for implementing shared counters and various data structures [22], and to analyze the performance differences between FAA and CAS.

Swap(*mem, reg) (SWP): it swaps the values in a memory location *mem and a register reg. We choose SWP due to its significance in, e.g., implementing locks [10].

Here, we focus on benchmarking the atomic assembly operations and we thus assume that each operation loads only one operand from the memory subsystem. The remaining operands are precomputed and stored in the respective registers. For CAS we also evaluate the variant with two operands fetched. Our strategy reflects many parallel codes and data structures where the arguments of the atomic function calls are constants or precomputed values; for example BFS traversals [23] or distributed hashtables [6].

The analyzed atomics have different *consensus numbers*, where *consensus* is the problem of agreeing on one value in the presence of many parties [10]. The consensus number of an operation `op`, denoted as `CN(op)`, is the maximum number of threads that can reach consensus with a wait-free algorithm that only uses reads, writes, and `op`. In this evaluation, we select both the operations that have smaller consensus numbers (`CN(SWP) = CN(FAA) = 2`) and the operation with a high consensus number (`CN(CAS)` = ∞) to analyze whether it has any performance implications.

3. Design of Benchmarks

Measuring the performance of atomics is non-trivial due to the complexity of deep memory hierarchies, various types of workloads with different caching patterns, and the richness of hardware mechanisms such as cache prefetchers that influence the performance results [8]. We now present the methodology that overcomes these challenges. We conduct:

Latency benchmarks: Here, pointer chasing is used to obtain the average latency of an atomic. This benchmark targets latency-constrained codes such as shared counters or synchronization variables used in parallel data structures.

Bandwidth benchmarks: Here, all the memory cells of a given buffer are accessed sequentially and the bandwidth is measured. While this part targets some bandwidth-intensive codes such as graph traversals [23], it also shows that the tested atomics do not enable any instruction-level parallelism (ILP) even if there are no dependencies between issued operations.

3.1 Relevant Parameters

We focus on the following parameters that impact the performance of atomics:

Type of atomic: we evaluate different atomic operations to cover a broad range of data structures and workloads (e.g., different types of concurrent queues may use either CAS [**?**] or FAA [**?**]). In addition, we illustrate the tradeoffs between the consensus number of atomics [**?**] and their performance. We evaluate CAS, SWP, and FAA.

Cache coherency state: we use cache lines in various CC states (M,E,S,O,I) to analyze the impact of the CC protocol on the performance of atomics.

Cache proximity: we place the accessed cache line in different caches to evaluate the impact of state-of-the-art deep cache hierarchies. The data accessed by a core can be in its local cache or in another core's cache located: on the same die, on a different die but on the same CPU, or on a different CPU.

Memory proximity: we use memories with different proximities to cover today's NUMA systems. We will refer to a memory that

can be accessed by a core without using a die-die interconnection as the *local memory* and anything else as the *remote memory*.

Thread count: we vary the number of threads accessing the same cache line to analyze the overheads due to contention.

Operand size: we evaluate operations that modify operands of various sizes to discover the most advantageous size to be used for shared counters or synchronization variables.

3.2 Structure of Benchmarks

The general structure of the benchmarks is similar to the structure described in Section 2.1 with the difference of measuring atomic instructions instead of reads or writes. CAS however is a special case which needs further adjustments. When the old value in the register (reg1) does not correspond to the value in memory (*mem), CAS fails and no memory location will be modified. However, when the old value reg1 is equal to *mem, CAS succeeds and there will be a write to memory. We investigate these cases separately.

CAS: Bandwidth Benchmarks In the bandwidth benchmarks for successful CAS, we fill the buffer with zeros and use zero as the old value (reg1). For unsuccessful CAS, we fill the buffer with the increasing byte values. When a CAS fails, reg1 is updated to *mem. This value will differ from the next one in the buffer, ensuring that all the issued CAS operations will fail.

CAS: Latency Benchmarks We measure the latency of the unsuccessful CAS by filling the buffer with the increasing values and comparing each new fetched value with the previous one. This ensures that each CAS fails. For CAS to be successful we need to know *mem in advance. With the pseudo random addresses this cannot be achieved without additional memory accesses that would interfere with the benchmarks. Instead, we use another approach: We fill the buffer with zeros, split it into equally sized chunks, and perform a predefined access pattern using the beginning of each chunk as the base address. If we benchmarked reads with this approach they would be executed in parallel because there is no data dependency between the reads, preventing the exact latency measurement. However, CAS affects the register containing the old value and that value also affects the outcome of the next operation so there is a data dependency. Thus, the instructions are serialized and their latency can be correctly measured.

CAS vs FAA vs SWP: Instruction Level Parallelism On all the tested systems the CAS assembly operation always modifies the same predefined register. Thus, the CPU cannot execute multiple CASes simultaneously because the result of one CAS affects the outcome of the next CAS. FAA and SWP however have only one explicit argument. Our bandwidth benchmarks avoid data dependencies between the instructions to allow parallel execution of FAA and SWP. We will later illustrate that the hardware implementation of each atomic still enforces fully serialized execution.

3.3 Interference from Hardware Mechanisms

There are several mechanisms that could introduce significant noise in the benchmarks; we turn them off where possible. First, we avoid TLB misses by using hugepages and filling the TLB with proper entries prior to the measurements. Second, we disable the respective mechanisms that affect the clock frequency; these are Turbo Boost, Enhanced Intel SpeedStep (EIST), and CPU C-states. Thus, the frequency of each core is always as specified in Table 1. Third, we turned off prefetchers (Hardware Prefetcher, Adjacent Cache Line Prefetch) to prevent false speedups in the latency benchmarks. In some of the systems (Ivy Bridge, Bulldozer) we could not influence the hardware configuration and we avoided prefetching by applying sparser access patterns. Finally, by switching off HyperThreading we make sure that any two cores visible to the programmer are also two physical cores.

4. Performance Model

We now introduce our performance model. We concretize the model by assuming that we model caching architectures that match the considered Intel and AMD systems (cf. Section 2.2 and Table 1). We will later (Section 5) validate the model and explain several differences between the predictions and the data that illustrate interesting architectural properties of the considered systems.

4.1 Latency

Each atomic fetches and modifies a given cache line ("read-modify-write"). We predict that an atomic first issues a read for ownership in order to fetch the respective cache line and invalidate the cache line copies in other caches. Then the operation is executed and the result is written in a modified state to the local L1 cache. We thus model the latency $\mathcal{L}$ of an atomic operation A executing with an operand from a cache line in a coherency state S as:

$$\mathcal{L}(A,S) = \mathcal{R}_O(S) + \mathcal{E}(A) + \mathcal{O} \tag{1}$$

A denotes the analyzed atomic; $A \in \{\mathsf{CAS}, \mathsf{FAA}, \mathsf{SWP}\}$. S denotes the coherency state; $S \in \{\mathsf{E}, \mathsf{M}, \mathsf{S}, \mathsf{O}\}$. $\mathcal{R}_O(S)$ is the latency of the read for ownership (reading a cache line in a coherency state S and invalidating other caches). $\mathcal{E}(A)$ is the latency of: locking a cache line, executing A by the CPU, and writing the operation result into a cache line in the coherency state M. As all other copies of the cache line are invalidated, this will be a write into L1 local to the core executing the instruction. Finally, $\mathcal{O}$ denotes additional overheads related to various proprietary optimizations of the coherence protocols that we describe in Section 5. We conjecture that the most dominant element of $\mathcal{L}(A,S)$ is $\mathcal{R}_O(S)$; a prediction supported by several studies illustrating high latencies of reads for ownership [8, 20, 21].

$\mathcal{R}_O(S)$ strongly depends on S and the location of the cache line. We start with modeling operations that access cache lines located on the same die as the requesting core.

4.1.1 On-die Accesses: E/M states

If S is E or M then there is a single copy of the related cache line and no invalidations will be issued. Thus, $\mathcal{R}_O(\mathsf{E})$ and $\mathcal{R}_O(\mathsf{M})$ will be equal to the latency of a simple read denoted as $\mathcal{R}$:

$$\mathcal{R}_O(\mathsf{E/M}) = \mathcal{R}(\mathsf{E/M}) \tag{2}$$

Private L1 and L2, shared L3 We first assume that each core has private L1 and L2 caches and there is a shared L3 across all the cores. Examples of such systems are the considered Intel Ivy Bridge and Intel Haswell configurations. We first denote the latency of reading a cache line by a core from a local L1, L2, and L3 cache as $\mathcal{R}_{L1,l}$, $\mathcal{R}_{L2,l}$, and $\mathcal{R}_{L3,l}$, respectively. Then, we have:

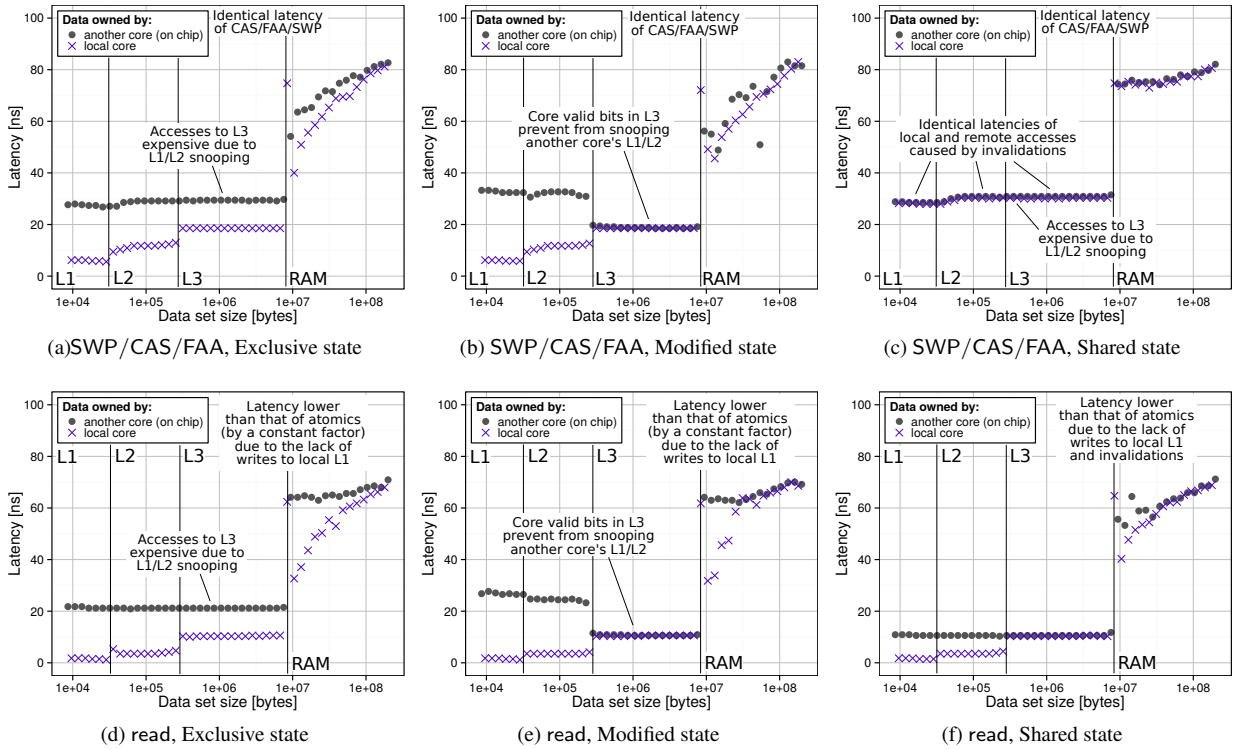
$$\mathcal{R}(\mathsf{E/M}) = \mathcal{R}_{L,l} \text{ iff the cache line is in L} \tag{3}$$

where $L \in \{L1, L2, L3\}$. We now model the latency of accessing a cache line in L1 or L2 of a different core. Here, we assume that the latency of transferring a cache line between L1 and L3 can be estimated as $\mathcal{R}_{L3,l} - \mathcal{R}_{L1,l}$. The total latency is increased by an additional cache line transfer from L3 to the requesting core:

$$\mathcal{R}(\mathsf{E/M}) = \mathcal{R}_{L3,l} + \mathcal{R}_{L3,l} - \mathcal{R}_{L1,l} \tag{4}$$

Private L1, shared L2 and L3 In some architectures (e.g., Bulldozer) L2 is shared. For such systems, if the cache line is in the L1 owned by a core that shares L2 with the requesting core, then:

$$\mathcal{R}(\mathsf{E/M}) = \mathcal{R}_{L2,l} + \mathcal{R}_{L2,l} - \mathcal{R}_{L1,l} \tag{5}$$



(a) SWP/CAS/FAA, Exclusive state (b) SWP/CAS/FAA, Modified state (c) SWP/CAS/FAA, Shared state

(d) read, Exclusive state (e) read, Modified state (f) read, Shared state

Figure 2: The comparison of the latency of CAS, FAA, SWP, and read on Haswell. The requesting core accesses its own cache lines (local) and cache lines of a different core from the same chip (on chip).

Private L1 and L2, no L3 Finally, Xeon Phi represents systems with private L1/L2 and no L3. Recent research [26] illustrated that the latency of a cache line transfer between any two cores on a Xeon Phi chip can be assumed identical. Thus, we have:

$$\mathcal{R}(\mathsf{E/M}) = \mathcal{R}_{L2,l} + \mathcal{R}_{L2,l} - \mathcal{R}_{L1,l} + \mathcal{H} \tag{6}$$

where $\mathcal{H}$ is the latency of the hop from the local L2 to the remote L2, including the overhead from accessing the directories maintained by the cache coherency protocol.

4.1.2 On-die Accesses: S/O states

If the cache line is in S or O, then the read for ownership invalidates its copies in other caches. Assuming there are N copies, we have:

$$\mathcal{R}_O(\mathsf{S/O}) = \mathcal{R}(\mathsf{S/O}) + \max_{i \in \{1,\ldots,N\}} \mathcal{L}_{inv,i} \tag{7}$$

where $\mathcal{L}_{inv,i}$ is the latency of invalidating the ith cache line. Here, we assume that multiple invalidations are executed in parallel, thus we take the maximum of the latencies. We also predict that $\mathcal{L}_{inv,i}$ should not significantly differ from $\mathcal{R}(\mathsf{E})$ (of the ith cache line), because both require invalidating private caches independent of data being cached there. Similarly, we approximate reads of S/O cache lines with reads of E cache lines:

$$\mathcal{R}_O(\mathsf{S/O}) = \mathcal{R}(\mathsf{E}) + \max_{i \in \{1,\ldots,N\}} \mathcal{R}_i(\mathsf{E}) \tag{8}$$

4.1.3 Off-die Accesses

The operations accessing cache lines located on a different die include an additional penalty from the underlying network (QPI on Intel and HT on AMD systems). Here, we assume a constant overhead $\mathcal{H}$ per one cache-to-cache hop (spanning two dies) that we add to the respective latency expressions from Section 4.1.2. The latency of accesses to the main memory $\mathcal{M}$ is modeled as a sum of the L3 miss and the overhead introduced by processing the request by the memory controller. For NUMA systems we also add $\mathcal{H}$ if necessary for an additional die-to-die hop. Finally, on Intel systems we also add $\mathcal{M}$ to each $\mathcal{R}(\mathsf{M})$ because such accesses require writebacks to memory; AMD prevents it with the O state.

4.2 Bandwidth

Here, we assume that the tested atomics always flush the write buffers and thus do not allow for ILP [12, 13]. Thus, the bandwidth $\mathcal{B}$ of an atomic A executing with an operand from a cache line in a coherency state S can be simply modeled as:

$$\mathcal{B}(A,S) = \mathcal{C}_{size}/\mathcal{L}(A,S) \tag{9}$$

where $\mathcal{C}_{size}$ is the cache line size. This model assumes that each atomic modifies a different cache line. In the case where the continuous memory block is accessed sequentially and thus each cache line is hit multiple times, we have:

$$\mathcal{B}(A,S) = \frac{\mathcal{N}}{\mathcal{L}(A,S) + (\mathcal{N}-1)\cdot(\mathcal{R}_{L1,l} + \mathcal{E}(A))} \cdot \mathcal{C}_{size} \tag{10}$$

where O_{size} is the operand size and $\mathcal{N} = \mathcal{C}_{size}/\mathcal{O}_{size}$ is the number of operands that fit into a cache line. Eq. (10) is valid for Intel systems. L1 on AMD is write-through, thus $\mathcal{R}_{L2,l}$ would replace $\mathcal{R}_{L1,l}$:

$$\mathcal{B}(A,C,S) = \frac{\mathcal{N}}{\mathcal{L}(A,C,S) + (\mathcal{N}-1)\cdot\mathcal{L}_{R,L2,l}} \cdot \mathcal{C}_{size} \tag{11}$$

5. Performance Analysis

We now illustrate our main results and provide several surprising insights into the performance of the tested atomics. We exclude the results that show similar performance trends; they are all included in an extended technical report[1]. We use the results to validate the model. Here, we first calculate the median values of the parameters from Section 4. The obtained numbers can be found in Table 2. We omit the model lines for clarity of plots and we discuss each case where the differences between the model and the data exceed 10% of the normalized root mean square error (NRMSE) defined as:

$$\text{NRMSE} = \frac{1}{\bar{x}}\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\hat{x}_i - x_i)^2} \tag{12}$$

where $\hat{x}_i$ are predictions, x_i are the observed data points, and $\bar{x}$ is the mean of the observed values.

Parameter:	Haswell	Ivy Bridge	Bulldozer	Xeon Phi
$\mathcal{R}_{L1,l}$	1.17	1.8	5.2	2.4
$\mathcal{R}_{L2,l}$	3.5	3.7	8.8	19.4
$\mathcal{R}_{L3,l}$	10.3	14.5	30	-
$\mathcal{H}$	-	66	62	161.2
$\mathcal{M}$	65	80	75	340
$\mathcal{E}$(CAS, .)	4.7	4.8	25	12.4
$\mathcal{E}$(FAA, .)	5.6	5.9	25	2.4
$\mathcal{E}$(SWP, .)	5.6	5.9	25	3.1

Table 2: The model parameters (all numbers are in nanoseconds).

The overhead term $\mathcal{O}$ (see Eq. (1)) depends on the operation type, the coherency state, the accessed cache line, and the architecture; we present a selection of $\mathcal{O}$ values in Table **??**. The available vendor documentations and manuals prevent the definite explanation of the reasons behind the variability of $\mathcal{O}$ [1, 12, 13]. We conjecture that the reasons may include: proprietary undocumented optimizations, variable locking overheads, or different snooping techniques (e.g., whether or not the snoop request bypasses the targeted core).

	Local			Remote		
	L1	L2	L3	L1	L2	L3
E state	0	3.8	3.5	3	5	5
M state	0	3.8	3.5	9	8	5
S state	3	1.4	-4	-15	-14	-12

Table 3: The O term for Haswell.

5.1 Latency

First, we present a selection of the latency results. We compare CAS, FAA, and SWP that access cache lines in the E, M, S and O states. We exclude the F state from the Intel analysis as it starts to affect the performance when more than two CPUs are used [20] while our Intel testbeds host at most two CPUs. We illustrate the results for unsuccessful CAS; successful CAS follows similar performance patterns. Finally, the latency of reads (read) is also plotted for a baseline comparison with a simple memory access.

5.1.1 Intel Haswell and Ivy Bridge

We illustrate the latency results of the Intel systems in Figures 2 and 3. Atomics are consistently slower than reads by ≈5-10ns on both systems for the E/M states (cf. Figures 2a and 2d). From this we conjecture that atomics trigger a read for ownership and the latency difference between atomics and simple reads stems from

[1] http://spcl.inf.ethz.ch/Research/Parallel_Programming/Atomics/

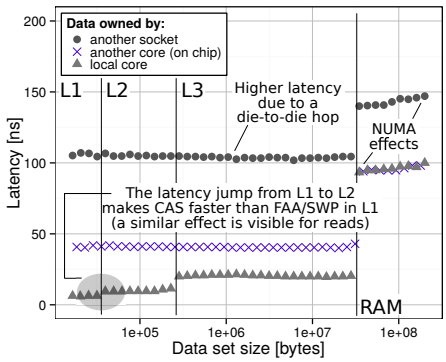


Figure 3: The analysis of the CAS latency (Exclusive state) on Ivy Bridge and the comparison to FAA/SWP. The requesting core accesses its own cache lines (local), cache lines of a different core from the same chip (on chip), and cache lines of a core from a different socket (another socket).

$\mathcal{E}$, as predicted by Eq. (1). The desired cache line is read into the private cache of the core and all its copies are invalidated. For cache lines in the E and M states a read for ownership has the same latency as a read since the line is only present in one cache, requiring no invalidations. The difference in latency impacts the performance of local L1 accesses where the read latency is ≈1-2ns (see Figure 2d). It does not significantly influence accesses to remote caches or memory where latencies are >60ns. *As predicted by our model and contrary to the common view, CAS has the same latency as FAA/SWP*, except for the E and M states on Ivy Bridge, where the latency of CAS accessing L1 is consistently (by ≈2-3ns) *lower* than that of FAA/SWP (see Figure 3). We attribute this effect to an optimization in the structure of L1 that detects that no modification will be applied to a cache line, reducing the latency.

In the S/E states executing an atomic on the data held by a different core (on the same CPU) is not influenced by the data location (L1, L2 or L3); see Fig. 2a, 2c, 2d, 3. The data is evicted silently, with neither writebacks nor updating the core valid bit in L3. Thus, all the accesses snoop L1/L2, making the latency identical (as modeled by Eq. (8)).

M cache lines are written back when evicted updating the core valid bits. Thus, there is no invalidation when reading an M line in L3 that is not present in any local cache. This explains why M lines have lower latency in L3 than E lines; cf. Figures 2a and 2b.

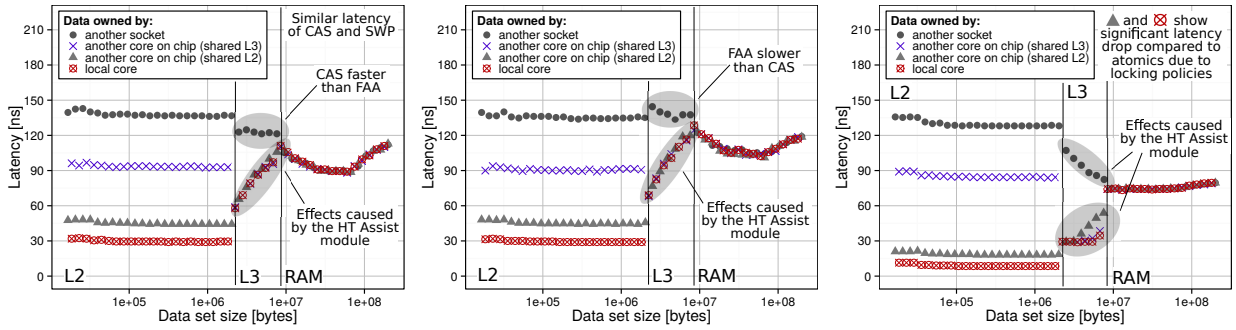
Remote accesses in the M/E states are by ≈50ns slower than that of another core on the same CPU due to $\mathcal{H}$; see Fig. 3. For modified cache lines the latency is different for L3 because the MESIF protocol does not allow for dirty sharing so the data has to be written to memory incurring $\mathcal{M}$.

In our latency benchmarks CAS does not write to L1. It is not necessary to invalidate cache lines sharing the data when performing unsuccessful CAS. The results indicate that the Intel architectures do not take advantage of that because a read for ownership might be issued in any case.

5.1.2 AMD Bulldozer

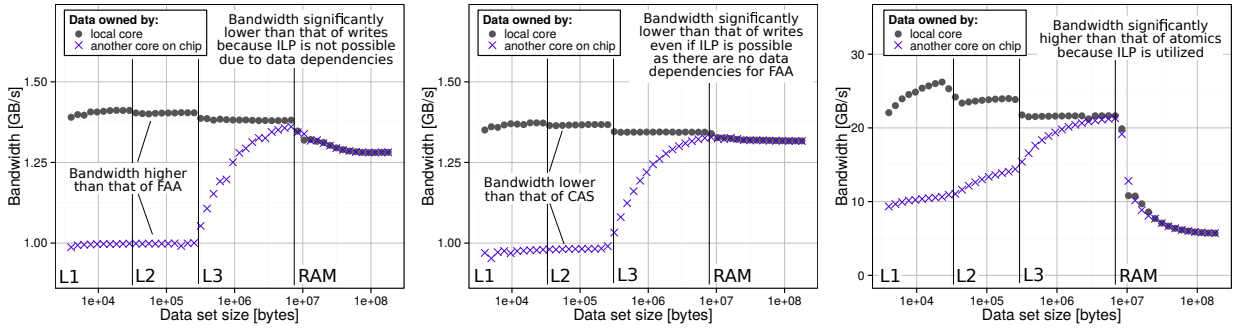
We illustrate the latency results of AMD Bulldozer[2] in Figure 4. Atomics are again slower than reads in each case. Yet, the difference between reads and CAS/FAA is not the same for all the cache levels. CAS and FAA take ≈8ns longer than reads into the cache of a different core. Yet, for the local cache they consistently take

[2] For AMD, we had to eliminate the effects from AMD hardware prefetchers that always request multiple consecutive cachelines in case of an L2 miss. For this, we had to increase the minimal distance between the accessed memory addresses. As the size of L1 in Bulldozer is only 16KB, this prevented obtaining results for L1. Identical effects were observed by Molka et al. [20].



(a) CAS/SWP, Exclusive state (b) FAA, Exclusive state (c) read, Exclusive state

Figure 4: The latency comparison of CAS/FAA/SWP/read on Bulldozer. The requesting core accesses its own cache lines (local), cache lines of different cores that share L2 and L3 with the requesting core (on chip, shared L2 and L3, respectively), and cache lines of a core from a different socket (other socket).



(a) CAS, Modified state (b) FAA, Modified state (c) write, Modified state

Figure 5: The comparison of the bandwidth of CAS, FAA, and writes on Haswell. The requesting core accesses its own cache lines (local) and cache lines of a different core from the same chip (on chip).

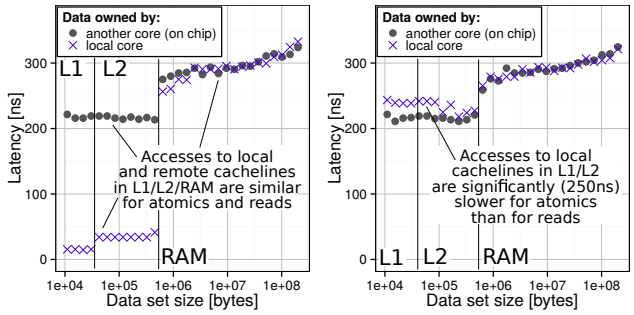
≈20ns longer than respective reads (cf. Figures 4a/4b and 4c). We attribute this surprising result to variable overheads related to $\mathcal{O}$.

The L3 results indicate that the latency is growing with the increasing data block size. We conjecture that this effect is caused by the HT Assist, a unit that uses a part of L3 and works as a filter for accesses to remote cores [1]. The HT Assist module causes some accesses to L3 to miss and thus to incur higher latencies.

Both the S and the O states follow similar performance patterns (we exclude the plots due to space constraints). The latencies of atomics to shared data in the L2 of the requesting core are similar independent of which cores contain the data; they are dominated by $\mathcal{H}$ (additional ≈62ns). Bulldozer's L3 is not inclusive and does not have core valid bits. Thus, L3 cannot determine whether the data is in the L1 or L2 of a different core entailing an invalidation broadcast. This broadcast has to reach caches on a remote CPU, generating very high latencies.

5.1.3 Intel Xeon Phi

Finally, we discuss the latency results for Xeon Phi, see Figure 6. The performance patterns are very similar across all the tested operations and coherency states as predicted by the model; here we illustrate CAS. Similarly to other Intel and AMD systems, atomics introduce significant overheads over reads for the S state (≈250ns for L1 accesses). Yet, contrarily to Haswell, Ivy Bridge, and Bulldozer, CAS is slower than FAA (≈10ns for local L1 and ≈15ns for remote L1 accesses) while FAA is slower than read (≈2ns for local L1 and ≈5ns for remote L1 accesses).



(a) CAS, M/E state (b) CAS, S state

Figure 6: The comparison of the latency of CAS on Xeon Phi. The requesting core accesses its own cache lines (local) and cache lines of a different core from the same chip (on chip).

This minor latency differences in accessing cache lines owned by different cores are due to the design of the Xeon Phi ring-bus: each of these two accesses requires checking different cache directories. This is consistent with previous results for memory accesses as observed by other studies [26].

5.1.4 Discussion & Insights

Our analysis provides novel insights. It turns out that, contrary to the common view [22], the latency of CAS, FAA, and SWP is in most cases identical and sometimes (L1 on Haswell and L3/memory on AMD) CAS is *faster* than FAA. Thus, atomics with different consensus numbers still entail similar overheads. As we will show in Section 5.5, additional overheads in CAS are due to fetching an additional argument from caches or memory.

The results indicate the correctness of the model assumptions and predictions. The only significant deviations are caused by factors not directly related to the cache coherency protocol (TLB misses) and the overheads from the proprietary HT Assist module on AMD Bulldozer. We also observe minor (<10%) variations in the latencies to remote caches caused by system noise.

The analysis suggests several potential improvements for the hardware implementation of atomics. For example, we show that unsuccessful CASes invalidate the copies of fetched cache lines entailing significant overheads. Yet, such operations do not modify the fetched cache line, making the invalidations unnecessary. We conjecture that this strategy incorporates the pipelining of atomics, thus requiring the invalidations. Another potential strategy would issue invalidations after CASes succeed. As unsuccessful CASes usually constitute a crucial part of all the issued CASes in various parallel designs [22], this might accelerate some workloads.

5.2 Bandwidth

We now analyze a selection of the bandwidth results. Due to space constraints we illustrate the Haswell results for the M state (Figure 5) and only briefly discuss other testbeds. Here, we compare atomics to writes. Our goal is to show that atomics do not use ILP even if no dependencies between successive operations exist.

Similarly to latency, the bandwidth results for Haswell indicate that CAS is *comparable or more efficient* than FAA (≈0.04 GB/s). Moreover, the bandwidth is larger in higher level caches (for the E/M cache lines). Yet, the differences between the levels are not significant (≈0.05 GB/s) as only the first access to each line is affected by cache proximity. Bandwidth (to L3) for the E lines is lower than for the M lines due to to the silent eviction of the former.

In each testbed the bandwidth of atomics is ≈5-30x lower than that of writes because the latter utilize ILP. Yet, in our design (see Section 3.2) we specifically enabled the possibility of parallel execution of FAA/SWP. We conjecture that the hardware implementation of atomics prevents ILP, limiting performance.

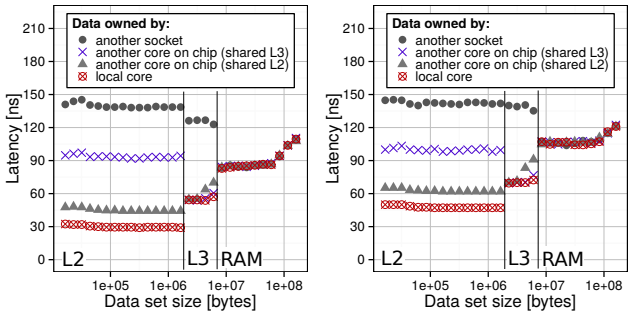
5.2.1 Discussion & Insights

Significantly lower bandwidth of atomics (than that of writes) is due to the differences in using write buffers. Cores store to their write buffers and continue executing further instructions before the previous writes actually reach the cache (which can take >100ns as our results indicate). The buffer might merge consecutive writes increasing bandwidth. On the contrary, atomic operations cause the write buffers to be drained. That means that every atomic is affecting the cache directly without being merged or buffered.

In addition, our results indicate that atomics do not allow for ILP whatsoever. Relaxing this restriction in some cases (e.g., for the independent executions of FAA or SWP) could significantly improve the bandwidth.

5.3 Operand Size

CAS comes with several flavors that differ in the size of the operands. We analyze variants that use 64 and 128 bits. The tested Intel systems provide identical latency in each case. On the contrary, AMD Bulldozer has lower latency when using 64 bits, see Figure 7. The latency difference is insignificant (≈5ns) when accessing cache of a core that does not share L2 with the requesting core and close to 20ns for other caches and memory. Using CAS



(a) CAS (64 bits) (b) CAS (128 bits)

Figure 7: The comparison of the latency of CAS using operands of 64 and 128 bits in size (AMD Bulldozer, the M state).

that operates on 64bit operands would thus be desirable in latency-constrained applications running on AMD Bulldozer.

5.4 Contention

We now evaluate the effect of concurrent threads accessing the same cache line (using writes and atomics) on the tested manycore systems (Ivy Bridge, Xeon Phi, Bulldozer); see Figure 8. This benchmark targets the codes with highly contended shared counters and synchronization variables.

The bandwidth of writes on Ivy Bridge has almost 100GB/s with eight cores and is growing steadily with the thread count. These numbers are very close to the accumulated non-contended bandwidth. We observe a similar effect on Haswell and thus we conjecture that both architectures detect that issued operations access the same cache line in an arbitrary order, annihilating the need for the actual execution of all the writes. Contrarily to other Intel systems, adding more threads on Xeon Phi quickly decreases the bandwidth until it converges to ≈708 MB/s (CAS), ≈730 MB/s (FAA) and ≈2960 MB/s (writes).

Bulldozer also suffers from the contention. A single thread reaches the highest bandwidth but additional threads (up to eight) decrease the performance. Beyond this point the bandwidth increases steadily, similarly to Ivy Bridge.

We conclude that all the considered architectures have significantly lower bandwidth in a contended execution of atomic operations than in a non-contended case. This may constitute a performance limitation in state-of-the-art multi- and manycore designs with massive thread-level parallelism.

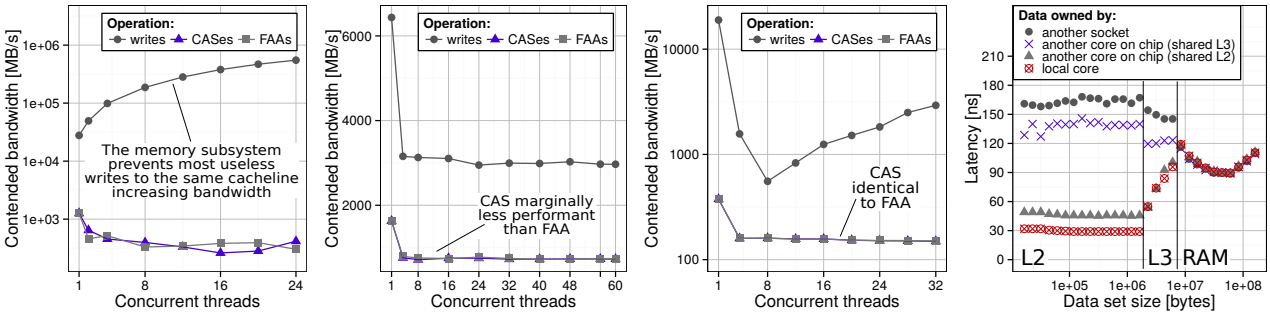
5.5 Number of Operands Fetched

Here, we show how the performance of CAS changes when two operands are fetched from the memory subsystem. We analyze the latency results for Bulldozer, see Figure 8d for the E state. It turns out that additional reads and invalidations impact the latency only marginally because of the pipelined execution of CASes/reads, requiring additional ≈2-4ns and ≈15-30ns for local and remote accesses, respectively.

Surprisingly, the latency of accessing M cache lines is not affected and is similar to the one from Figure 4a. This effect is caused by the proprietary AMD optimization called the *MuW* state [15]. It immediately invalidates the accessed M cache line and allows the requesting core to modify it without further actions, limiting remote invalidations and improving the performance even further.

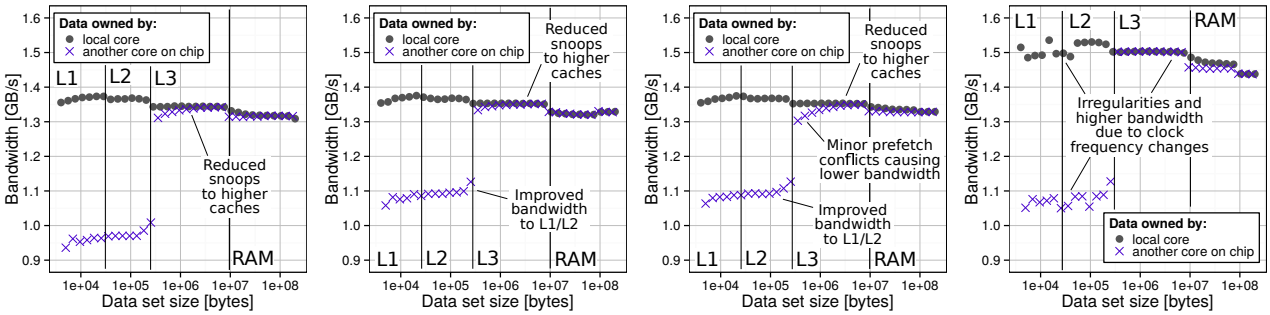
5.6 Prefetchers and Other Mechanisms

State-of-the-art architectures host different mechanisms that impact the CPU performance. For example, Haswell deploys Hardware Prefetcher (prefetching data/instructions after successive



(a) Ivy Bridge (b) Xeon Phi (c) Bulldozer (d) Bulldozer

Figure 8: Figures 8a-8c illustrate the comparison of the contended bandwidth of CAS/FAA/writes on Ivy Bridge, Bulldozer, and Xeon Phi. Figure 8d shows the latency of CAS that fetches two operands from the memory subsystem on Bulldozer (Exclusive state).



(a) Hardware Prefetcher. (b) Adjacent Cache Line Prefetcher. (c) Both prefetchers enabled. (d) Turbo Boost, EIST, C States.

Figure 9: The effect on the bandwidth of FAA (accessing cache lines in the Modified state) coming from prefetchers and various power efficiency and acceleration mechanisms (Turbo Boost, EIST, C States) deployed in Intel Haswell.

L3 misses or after detecting cache hit patterns), Adjacent Cache Line Prefetcher (unconditional prefetching of two additional cache lines), and several mechanisms that may affect the clock frequency and power efficiency (Turbo Boost, EIST, and C States). We now illustrate how these mechanisms impact the latency and bandwidth of atomics. We select Haswell as the testbed and we skip the latency results because they are only marginally (≈1% difference) affected. The bandwidth results are illustrated in Figure 9. Any of the prefetchers improves bandwidth for L3 cache accesses by reducing the effect of snooping (improvement up to ≈0.3 GB/s). Interestingly, if both are enabled, they negligibly conflict with each other reducing bandwidth to L3. Adjacent Cache Line Prefetcher additionally accelerates atomics to L1/L2 (up to ≈0.135 GB/s). Turbo Boost, EIST, and C States impact the clock frequency and thus both introduce irregularities in the results and improve the bandwidth of L3, RAM, and remote L1/L2 accesses by ≈0.15 GB/s.

6. Discussion

Our model and benchmarks provide insights into the latency and bandwidth of atomics; they also identify various performance issues. We now proceed to discuss how the insights can simplify parallel programming. Then, we propose several solutions to the identified performance issues.

6.1 Simplifying Parallel Programming

Our analysis illustrates that all atomics have comparable latency and bandwidth. We argue that the only significant difference between atomics that matters for performance is the semantics (e.g., CAS introduces the notion of wasted work). This conclusion reduces the complexity of designing parallel lock-free algorithms.

As an example, consider a parallel synchronous Breadth First Search (BFS) graph traversal included in the well-known Graph500 benchmark [23]. A key part of this algorithm is a concurrent array called `bfs_tree` that is constructed during the traversal. Initially, each array cell equals -1. After the traversal, given a graph with n vertices labeled from 0 to $n-1$, `bfs_tree[v]` contains the label of the parent of vertex v as determined by the traversal.

Now, concurrent accesses to the array cells can be performed with CAS, SWP, or FAA. Our model and analysis clearly indicate that the latency and bandwidth of these actions are almost identical. Thus, the only significant factor is the semantics. As FAA always succeeds, it might happen that some threads updating the same cell would both increase the value of the cell. This would require a complex scheme in which the effects of some of the issued operations would be reverted. On the contrary, CAS and SWP can be used to design simpler protocols. However, as CAS introduces wasted work, we expect it to generate some additional overheads. We illustrate the results of a BFS traversal (4 concurrent threads) using both of these actions in Figure 10. A traversal is performed on Kronecker graphs [16] that model heavy-tailed real-world graphs. The largest graphs fill the whole available memory. The results illustrate that SWP results in more edges traversed per second.

6.2 Addressing the Shortages of the Tested Architectures

Designing feasible solutions to the identified issues would require access to the details of the benchmarked architectures. Unfortu-

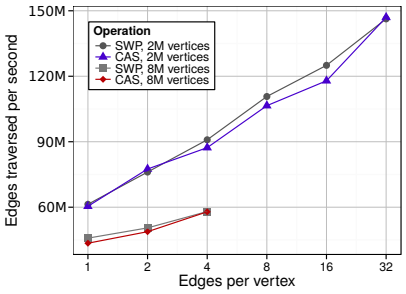


Figure 10: A BFS traversal implemented with SWP and CAS.

nately, such details are Intellectual Property (IP) of AMD/Intel that are beyond our access. Thus, we exclude microarchitectural details and we focus on general schemes that we believe could be easily incorporated into current architectures. The issue of insufficient information on microarchitectural features from hardware vendors has already been indicated by Mytkowicz et al. [24].

On Bulldozer, atomics accessing O/S cachelines always trigger invalidations to remote dies because the architecture lacks a method to track which caches share a cacheline. We now discuss two alterations to the Bulldozer design; each would prevent unnecessary invalidations to remote dies.

6.2.1 Extending MOESI

We extend MOESI with two new cache coherency states named **O**wned-**L**ocal (OL) and **S**hared-**L**ocal (SL). When a cacheline in the E state is read from a core on the same die it enters the SL state in both caches, rather than the S state. Similarly, when a cacheline in the M state is read by a core on the same die, it enters the OL/SL state instead of O/S. The SL and OL states indicate that the cacheline is only cached by the local die. Writing to an OL/SL cacheline requires no invalidations to a remote die. However, when an OL/SL cacheline is read by a core on a remote die, all the copies of that cacheline will transition from SL to S or from OL to O indicating that remote invalidations will be necessary when modifying the cache line.

The introduced states prevent unnecessary invalidations when writing to cachelines shared by cores on the same die. A potential disadvantage is that reading an SL or OL cacheline from a remote die might be slightly slower due to multiple transitions. However, this could be addressed with a careful overlap of the cacheline transfer and the transitions on a die.

6.2.2 Extending HT Assist

One could also eliminate unnecessary remote invalidations on AMD by using HT Assist to track S and O cachelines that are only present on one die. For this, a portion of L3 on each die would be dedicated to HT Assist to track the cachelines that most recently changed to the S or O state. Upon a read of an S/O cacheline (issued by a remote core), the respective entry in HT Assist would be removed. Upon a write (by a remote core), the HT Assist would be probed to determine whether to issue remote invalidations. Probing HT Assist does not increase the latency of local writes as L3 is consulted in any case.

6.2.3 Enabling ILP for Atomics

Atomics act as barriers and they require write buffers to be drained before they are executed [12, 13]. To enable ILP, an additional set of *relaxed* atomics could be introduced. For this, an instruction prefix called FastLock could be added to the instruction set. The FastLock prefix would enable reordering issued atomic operations as long as non-overlapping memory regions are affected. The non-relaxed semantics would be available with the original lock prefix.

7. Related Work

To the best of our knowledge, there exists no detailed performance analysis of atomic operations. A brief discussion that compares the contention of Compare-And-Swap and Fetch-And-Add can be found in the first part of the work by Morrison et al. [22]. This work uses the comparison to motivate the proposed parallel queue that extensively utilizes Fetch-And-Add. It differs from the study in this paper as it only illustrates the lower performance of CAS caused by the semantics that introduce wasted work. Another work by David et al. [4] analyzes the performance of atomics. Yet, this study targets a broad range of synchronization mechanisms and does not provide an in-depth analysis of both the latency and bandwidth of atomics in the context of complex multilevel memory hierarchies.

A methodology for benchmarking the latency and bandwidth of reads and writes accessing different levels of the caching hierarchy in the NUMA systems was conducted by Molka et al. [21]. A comparison of the performance of memory accesses on Intel Nehalem and AMD Shanghai was performed by Molka et al. [8]; a similar study targets the AMD Bulldozer and Intel Sandy Bridge microarchitectures [20]. Other analyses on the performance of the memory subsystems include the work by Babka et al. [2], Pend et al. [25], and Hristea et al. [11]. Our works differs from these studies as it specifically targets atomic operations, providing several insights into the performance relationships between atomics and the utilized caching hierarchy.

There exist numerous works proposing concurrent codes and data structures that use atomics for synchronization. Examples include a queue by Morrison at el. [22], a hierarchical lock by Luchangco et al. [18], and a queue by Michael and Scott [19]. Many fundamental structures and designs can be found in a book by Herlihy and Shavit [10].

Finally, the considered architectures and cache coherency protocols are extensively described in various manuals and papers [1, 7, 12–14, 28]. Several performance models targeting on-chip communication have been introduced, for example a model by Ramos and Hoefler [26]. The model proposed in this paper differs from that work because it specifically targets latency and bandwidth of atomic operations in the onnode environment.

8. Conclusion

Atomic operations are used in numerous parallel data structures, applications, and libraries. Yet, there exists no evaluation that would illustrate tradeoffs and relationships between the performance of atomics and various characteristics of multi- and many-core environments.

In this work we propose a performance model and provide a detailed evaluation of the latency and bandwidth of several atomic operations (Compare-And-Swap, Fetch-And-Add, Swap) that validates the model. The selected atomics are widely utilized in various parallel codes such as graph traversals, shared counters, spinlocks, and numerous data structures. Our performance insights include the observation that CAS and FAA have in principle identical latency and the only difference is related to the number of operands to be fetched and the semantics of CAS that introduce the notion of the "wasted work". Another insight is that the atomics prevent any instruction level parallelism, significantly limiting the bandwidth (up to 30x in comparison with simple writes), even if there are no dependencies between the successive operations. Our analysis can thus be used for designing more efficient parallel systems.

The results also indicate several potential improvements in the design of the caching hierarchy. For example, the AMD Bulldozer architecture limits performance with invalidations issued to remote

CPUs even if the respective cache line is stored only in local caches. Eliminating such invalidations would significantly accelerate atomic operations accessing cache lines in the shared state. Here, we discuss two general solutions to this problem.

Finally, we illustrate that our analysis simplifies the design of parallel algorithms. Our study and data can be used by architects and engineers to develop more performant memory subsystems that would offer even higher speedups for parallel workloads.

References

[1] AMD Corporation. Software Optimization Guide for AMD Family 15h Processors, January 2014.

[2] V. Babka and P. Tůma. Investigating cache parameters of x86 family processors. In *Proc. of the 2009 SPEC Bench. Work. on Comp. Perf. Eval. and Bench.*, pages 77–96, 2009.

[3] M. Butler. AMD "Bulldozer" Core-a new approach to multithreaded compute performance for maximum efficiency and throughput. In *IEEE HotChips Symp. on High-Perf. Chips (HotChips 2010)*, 2010.

[4] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. In *ACM Symp. on Op. Sys. Prin.*, SOSP '13, pages 33–48, 2013.

[5] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. In *Proc. of Intl. Symp. Comp. Arch.*, ISCA '11, pages 365–376, 2011.

[6] R. Gerstenberger, M. Besta, and T. Hoefler. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proc. of ACM/IEEE Supercomputing*, SC '13, pages 53:1–53:12, 2013.

[7] J. R. Goodman and H. H. Hum. Forward state for use in cache coherency in a multiprocessor system, 2005. US Patent 6,922,756.

[8] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *Proc. of the IEEE/ACM Intl. Symp. on Microarch.*, MICRO 42, pages 413–422, 2009.

[9] T. L. Harris. A Pragmatic Implementation of Non-blocking Linked-Lists. In *Intl. Conf. on Dist. Comp.*, DISC '01, pages 300–314, 2001.

[10] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[11] C. Hristea, D. Lenoski, and J. Keen. Measuring Memory Hierarchy Performance of Cache-coherent Multiprocessors Using Micro Benchmarks. In *ACM/IEEE Supercomputing*, SC '97, pages 1–12, 1997.

[12] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual, September 2014.

[13] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, September 2014.

[14] T. Jain and T. Agrawal. The Haswell microarchitecture–4th generation processor. *Intl. J. of Comp. Sc. and Inf. Tech.*, 4(3):477–480, 2013.

[15] K. M. Lepak, V. Kalyanasundharam, W. A. Hughes, B. Tsien, and G. D. Donley. Method and apparatus for accelerated shared data migration, May 20 2014. US Patent 8,732,410.

[16] J. Leskovec, D. Chakrabarti, J. Kleinberg, C. Faloutsos, and Z. Ghahramani. Kronecker Graphs: An Approach to Modeling Networks. *J. Mach. Learn. Res.*, 11:985–1042, Mar. 2010.

[17] X. Liao, L. Xiao, C. Yang, and Y. Lu. Milkyway-2 supercomputer: system and application. *Fron. of Comp. Science*, 8(3):345–356, 2014.

[18] V. Luchangco, D. Nussbaum, and N. Shavit. A Hierarchical CLH Queue Lock. In *Intl. Conf. on Par. Proc.*, Euro-Par'06, pages 801–810, 2006.

[19] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proc. of ACM Symp. on Prin. of Dist. Comp.*, PODC '96, pages 267–275, 1996.

[20] D. Molka, D. Hackenberg, and R. Schöne. Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. In *Work. on Mem. Syst. Perf. and Corr.*, MSPC '14, pages 4:1–4:10, 2014.

[21] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *Proc of Intl. Conf. on Par. Arch. and Comp. Tech.*, PACT '09, pages 261–270, 2009.

[22] A. Morrison and Y. Afek. Fast Concurrent Queues for x86 Processors. In *Proc of the ACM SIGPLAN Symp. on Prin. and Prac. of Par. Prog.*, PPoPP '13, pages 103–112, 2013.

[23] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang. Introducing the graph 500. *Cray User's Group (CUG)*, 2010.

[24] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! In *Proc. of the Intl. Conf. on Arch. Sup. for Prog. Lang. and Op. Sys.*, ASPLOS XIV, pages 265–276, 2009.

[25] L. Peng, J.-K. Peir, T. K. Prakash, C. Staelin, Y.-K. Chen, and D. Koppelman. Memory Hierarchy Performance Measurement of Commercial Dual-core Desktop Processors. *J. Syst. Archit.*, 54(8):816–828, Aug. 2008.

[26] S. Ramos and T. Hoefler. Modeling Communication in Cache-coherent SMP Systems: A Case-study with Xeon Phi. In *Intl. Symp. on High-perf. Par. Dist. Comp.*, HPDC '13, pages 97–108, 2013.

[27] D. Slogsnat, A. Giese, and U. Brüning. A Versatile, Low Latency HyperTransport Core. In *Proc. of the ACM/SIGDA Intl. Symp. on Field Prog. Gate Arr.*, FPGA '07, pages 45–52, 2007.

[28] T. Suh, D. M. Blough, and H.-H. Lee. Supporting cache coherence in heterogeneous multiprocessor systems. In *Des., Aut., and Test in Eur. Conf. and Exhib.*, volume 2, pages 1150–1155. IEEE, 2004.

[29] C. T. Vaughan. Application characteristics and performance on a Cray XE6. In *Proc. 53th Cray User Group Meeting*, 2011.

[30] R. Wolf. Nasa Pleiades Infiniband Communications Network, 2009. Intl. ACM Symp. on High Perf. Dist. Comp.

[31] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing. In *Proc. of the ACM/IEEE Supercomputing*, SC '13, pages 19:1–19:11, 2013.