ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Modelling and Evaluating Performance of Atomic Operations

Hermann Schweizer
hermannschweizer@hotmail.com

January 11 2015

Advisors: Prof. Dr. T. Hoefler, Maciej Besta
Department of Computer Science, ETH Zürich

Abstract

Atomic operations (atomics) such as Compare-and-Swap or Fetch-and-Add are ubiquitous in parallel programming for multi- and many-core architectures. 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 work 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, including Intel Haswell, Ivy Bridge, and AMD Bulldozer. The results illustrate surprising performance relationships between the considered atomics and architectural properties such as the coherence state of the accessed cache lines. For example, we show that the hardware implementation of atomics prevents any instruction-level parallelism even if there are no dependencies between the issued operations. Our insights unveil undocumented performance properties of the tested systems, enable more effective parallel programming, and accelerate data processing on various architectures deployed in both off-the-shelf machines and large compute systems.

Contents

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 Haswell with up to 18 cores on a chip installed in high-end servers [4], or AMD Bulldozer with 32 cores per node deployed in Cray XE6 machines [26]. Moreover, the number of cores on a chip is growing steadily and CPUs with hundreds and even thousands 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 [9].

Virtually all such architectures provide atomic operations for synchronization in parallel codes. Many of them (e.g., Test-and-Set) can be used to implement locks [11]. Others, e.g., Fetch-and-Add and Compare-and-Swap, enable constructing lock-free and wait-free algorithms and data structures that have stronger progress guarantees than lock-based codes [11].

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 [21]. 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 [10, 21]. Yet, to the best of our knowledge, no detailed model and benchmarks analyze the latency or bandwidth of the execution of the actual operations. 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 sockets connected with a HyperTransport (HT) link, each CPU consists of two dies, and each die provides one L3 cache and four L2 caches shared by eight cores [26]. It is unclear what the performance of different atomics is on such a system, what is the influence of the cache coherency state of the accessed cache line, how does the latency and bandwidth scale with the growing number of concurrent threads, what is the performance impact of mechanisms such as adjacent cache line prefetchers, and whether optimizations such as intruction-level parallelism are available for atomics.

In this work, 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 atomics (Compare-and-Swap, Fetch-and-

Add, Swap). Our results unveil undocumented architectural properties of the tested systems and can be used to design more performant codes in areas such as graph analytics or concurrent data structures. 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 the tested systems, and to suggest several improvements in the hardware implementation of respective atomics.

2 Background

We now present a general approach for benchmarking memory accesses (§ 2.1); we will later use and extend it to measure the performance of various synchronization mechanisms. Then, we present the evaluated architectures and synchronization mechanisms (§ 2.2, § 2.3). We finish with a discussion on the related cache coherency protocols (§ 2.4).

2.1 Benchmarking Memory Accesses

In our analysis we use and extend the X86membench infrastructure for benchmarking memory accesses [9], as well as the performance analysis tool BenchIT [23]. We use both the latency and the bandwidth benchmarks. 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 § 3). The TLB is warmed up and the data is placed in caches in the selected coherency state.

Synchronization & coordination: 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: A time stamp t_start is taken by each participating thread, a measurement is done, and the other time stamp t_end is taken.

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

Repetition: The previous phases are repeated multiple times for each analyzed buffer size. The fastest execution per buffer size is retained since any interference or overhead increases the execution time and thus the fastest execution is the least affected by interference or overhead [9].

2.2 Evaluated Architectures and Systems

In the analysis, we consider the following architectures and systems (consult Figure 1 and Table 1 for more details):

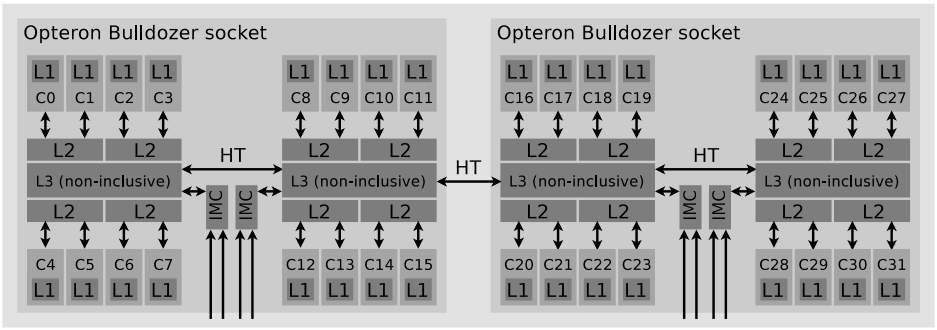
Haswell is an Intel state-of-the-art microarchitecture that offers sophisticated mechanisms such as hardware transactional memory (HTM) [28]. In our benchmarks we use a quadcore Haswell chip included in a commodity off-the-shelf server machine; see Figure 1c. The L1 and L2 caches are private to each 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 supercomputing systems such as Tianhe-2 [16] or NASA Pleiades [27]. Here, we evaluate an Ivy Bridge configuration installed in the Euler computer cluster that contains two 12-core CPUs connected with Quick Path Interconnect (QPI). The L1 and L2 caches 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.

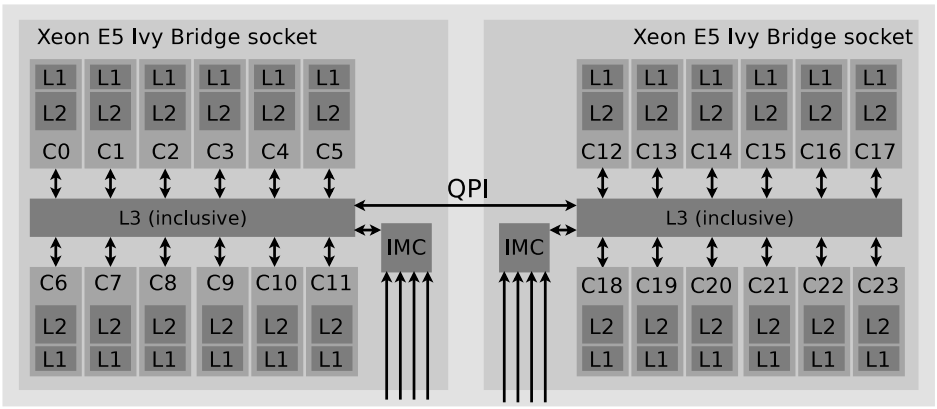
Bulldozer is an AMD microarchitecture designed to improve power efficiency for HPC applications [3]. Here, we evaluate a configuration included in the Cray XE6 Monte Rosa supercomputer [26]; see Figure 1a. A compute node contains two 16-core AMD Bulldozer Interlagos CPUs. Each CPU contains two 8-core dies that are connected with HyperTransport [24]. We selected this system to unveil differences between Intel and AMD systems and to analyze the effects coming from a particularly complex design with deep memory hierarchies with three cache levels, multiple CPUs, shared L2 caches, and multiple dies per CPU.

An important difference between the Intel and AMD systems is the structure of L3; we will later show that it significantly influences the performance of atomics. 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.

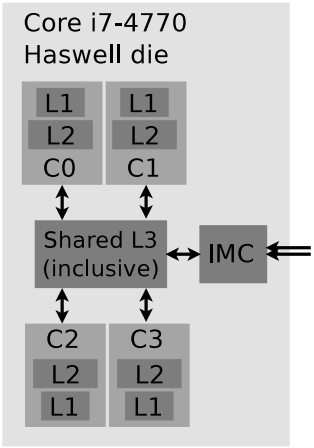
On the contrary, the L3 cache in AMD Bulldozer is neither exclusive nor inclusive: the presence of a cache line in L2 does not determine its presence in higher level caches. This will have a detrimental effect on the performance of atomics as we will illustrate in Section 5.



(a) AMD Bulldozer.



(b) Intel Ivy Bridge.



(c) Intel Haswell.

Figure 1: (§ 2.2) The illustration of the analyzed architectures.

	Architecture:	Haswell	Ivy Bridge	Bulldozer
Processor	Manufacturer	Intel	Intel	AMD
	CPU model	Core i7-4770	Xeon E5-2697v2	Opteron 6272
	Cores/CPU	4	12	16(2x8)
	CPUs	1	2	2
	Design	SMP	ccNUMA	ccNUMA
	Core frequency	3400 MHz	2700 MHz	2100 MHz
	Interconnect	-	2x QPI (8.0 GT/s)	4x HT 3.1 (6.4 GT/s)
Caches	Cache line size	64B	64B	64B
	L1 cache	32KB per core	32KB per core	16KB per core
	L1 Update policy	write back	write back	write through
	L2 cache	256KB per core	256KB per core	2MB per 2 cores
	L2 Update policy	write back	write back	write back
	L2 incl/excl:	neither	neither	neither
	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
Memory	Main memory	8GB	64GB	32GB
	Memory frequency	1600 MHz	1866 MHz	1600 MHz
	memory channels/CPU	1x dual channel	2x dual channel	2x dual channel
	Huge page size	2MB	2MB	2MB
Others	Linux kernel used	3.14-1	2.6.32	2.6.320
	Compiler	gcc 4.9.1	gcc 4.8.2	gcc 4.7.2
Assembly instructions	CAS	`Cmpxchg`	`Cmpxchg`	`Cmpxchg`
	FAD	`Xadd`	`Xadd`	`Xadd`
	SWP	`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).

2.3 Evaluated Synchronization Mechanisms

We now present the evaluated synchronization mechanisms:

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 [11] as well as various graph processing codes such Graph500 [22].

Fetch-and-Add(*mem, reg) (FAD): it fetches the value from a memory location *mem into a register reg and adds the previous value from reg to *mem. We selected FAD because of its importance for implementing shared counters and various data structures [21], and to analyze the performance differences between FAD and CAS.

Swap(*mem, reg) (SWP): it swaps the values in a memory location *mem and a register reg. SWP can be used to implement simple spinlocks [11].

Here, we focus on benchmarking the atomic assembly operations and we thus assume that each atomic fetches only one operand from the memory subsystem. The remaining operands are assumed to be precomputed and stored in respective registers. 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 [22] or distributed hashtables [7].

The analyzed atomics have different *consensus numbers*, where *consensus* is the problem of agreeing on one value in the presence of many parties [11]. 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(FAD) = 2`) and the operation with a high consensus number (`CN(CAS)` $= \infty$) to analyze whether it has any performance implications.

2.4 Related Cache Coherency Protocols

Here, we briefly discuss related cache coherency protocols.

MESI: This is the most common cache coherence protocol for write back caches. It tries to minimize the number of required memory accesses. Each cache line is in one of the following states: **M**odified, **E**xclusive, **S**hared, **I**nvalid. A Modified cache line is *dirty*; it differs from the value in main memory and any other cache. Before allowing another core to read this cache line, it must be written back to memory. A Exclusive cache line is only present in this cache and is clean meaning the data is identical to that in memory. A Shared cache line might be present in many caches and is clean. A Invalid cache line contains data which is inconsistent with the most recent write and is thus useless.

MESIF: When a shared cache line is requested by a core, in the MESI protocol this request might either be serviced by main memory or all the caches holding the data in Shared state. The MESIF protocol prevents such redundant data transfers by adding the **F**orward state which indicates that a cache should act as a designated responder for any requests for the given line.

MOESI: When a modified cache line is read by another core it needs to be written back to memory before it transitions into the shared state in both caches. The MOESI protocol prevents this write-back by introducing the **O**wned state which allows a dirty cache line to be shared. A cache line can only be owned by one cache and others might have that line in the shared state. When a cache line in owned state is written it must be broadcast to all caches containing it.

3 Design of Benchmarks

Measuring the performance of atomics is non-trivial due to the growing complexity of deep memory hierarchies, various types of workloads with different caching patterns, and the richness of hardware mechanisms such as cache prefetchers that may interfere with the performance results [9]. We now present the methodology that overcomes these challenges. It will enable gathering results that we will use to improve the performance of data analytics and to assess various types of CPU designs to indicate which one brings the highest speedups in relevant workloads. We measure both the bandwidth and the latency:

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 measurement targets some bandwidth-intensive codes such as graph traversals [22], it also shows that atomics provided by the tested architectures do not enable any instruction-level parallelism (ILP) even if there are no dependencies between issued operations.

3.1 Relevant Parameters

The performance of atomics depends on many parameters. The most important ones are as follows:

Cache coherency state (CC-state): we use cache lines in various CC states to analyze the impact of the CC protocol on the performance of atomics. We consider the states of the relevant cache coherency protocols (M,E,S,O,I) but we skip F because our testbed configuration does not allow evaluating its influence.

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

Memory proximity (memory-p): we use memories with different proximities to cover today's NUMA memory hierarchies. We will refer to a memory

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

Thread count (threads): we vary the number of threads accessing the same cache line to illustrate the performance overheads which occur due to contention and to find out whether the growing numbers of cores can be directly translated to the higher performance of various types of codes.

Operand size (size): finally, we evaluate operations that modify operands of various sizes to discover the most advantageous size to be used for various use-cases such as shared counters or synchronization variables.

3.2 Structure of Benchmarks

The general structure of the benchmarks is similar to the structure described in § 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 if there is a difference in latency and bandwidth of those two cases.

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 it is necessary to know *mem in advance. With the pseudo random addresses this cannot be achieved without some additional memory accesses which would introduce unacceptable interference to the benchmark. Instead we decided on another approach to create a pseudo random accesses pattern where instructions are not executed in parallel. Here, we prepare the buffer to contain only zero values, 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. For CAS however this is not a problem because CAS effects the register containing the oldvalue and that value effects the outcome of the

next operation so there is a data dependency and the instructions can not be executed in parallel.

CAS vs FAD vs SWP: Instruction Level Parallelism On all the tested systems the assembly instruction for CAS does not allow the programmer to define the location of the first argument to CAS which is written by the instruction. For this reason the CPU cannot execute multiple CAS instructions simultaneously because the result of one CAS affects the outcome of the next CAS. FAD and SWP however have only one explicit argument. Our bandwidth benchmarks avoid data dependencies between the instructions to allow parallel execution of FAD and SWP. We will later illustrate that the hardware implementation of each atomic still enforces fully serialized execution.

3.3 Interference from Hardware Mechanisms

We identified several hardware mechanisms that could introduce significant noise in the benchmarks; we turned them off where possible. First, TLB misses have a significant effect on the performance. We avoid them by using hugepages (if available) and filling the TLB with the proper entries prior to executing the measurements. Second, there exist several mechanisms that affect the clock frequency; these are Turbo Boost, Enhanced Intel SpeedStep (EIST), and CPU C-states. By turning those off, the frequency of each core remains at the frequency specified in Table 1 at all times. Third, we turned off various prefetchers (Hardware Prefetcher, Adjecent Cache Line Prefetch) to prevent the CPU from prefetching cache lines and ultimately introducing false speedups to 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. § 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 C in a coherency state S as:

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

A denotes the analyzed atomic; $A \in \{\mathsf{CAS}, \mathsf{FAD}, \mathsf{SWP}\}$. S denotes the coherency state; $S \in \{\mathsf{E}, \mathsf{M}, \mathsf{S}, \mathsf{O}\}$. $\mathcal{R}_O(C, S)$ is the latency of the read for ownership (reading C in a coherency state S and invalidating other caches). $\mathcal{E}(A, C)$ is the latency of: locking the cache line C, executing A by the CPU, and writing the operation result into the cache line C in the coherency state M. As all other copies of C 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 § 5. We conjecture that the most dominant element of $\mathcal{L}(A, C, S)$ is $\mathcal{R}_O(C, S)$; a prediction supported by several studies illustrating high latencies of reads for ownership [9, 19, 20].

$\mathcal{R}_O(C, S)$ strongly depends on S and the location of C. We start with modelling 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 exists only one copy of C and the underlying protocol will not issue invalidations. Thus, $\mathcal{R}_O(C, \mathsf{E})$ and $\mathcal{R}_O(C, \mathsf{M})$ will be equal to the latency of a simple read denoted as $\mathcal{R}$:

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

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:

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

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:

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

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

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

4.1.2 On-die Accesses: S/O states

If C is in S or O, then the read for ownership invalidates the copies of C in other caches. Assuming there are N copies denoted as C_i ($i \in \{1, ..., N\}$), we have:

$$\mathcal{R}_O(C, \mathsf{S/O}) = \mathcal{R}(C, \mathsf{S/O}) + \max_{i \in \{1,...,N\}} \mathcal{L}_{inv}(C_i) \tag{6}$$

where $\mathcal{L}_{inv}(C_i)$ is the latency of invalidating C_i. Here, we assume that multiple invalidations are executed in parallel, thus we take the maximum

of the latencies. We also predict that $\mathcal{L}_{inv}(C_i)$ should not significantly differ from $\mathcal{R}(C_i, \mathsf{E})$, because both require invalidating private caches independent of data being cached there. Similarly, we approximate $\mathcal{R}(C, \mathsf{S/O})$ with $\mathcal{R}(C, \mathsf{E})$. Finally, we have:

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

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 die-to-die hop that we add to the respective latency expressions from § 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 IMC. In case of 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 $\mathcal{R}(C, \mathsf{M})$ because such accesses require writebacks to memory; AMD prevents it with the O state.

4.2 Bandwidth

Here, we utilize the notion that atomics provided by the analyzed systems always flush the write buffers and do not allow for ILP [13, 14]. Thus, the bandwidth $\mathcal{B}$ of an atomic A executing with an operand from a cache line C in a coherency state S can be simply modeled as:

$$\mathcal{B}(A, C, S) = 1/(\mathcal{L}(A, C, S)) \cdot \mathcal{C}_{size} \tag{8}$$

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 mutiple times, we have:

$$\mathcal{B}(A, C, S) = \frac{\mathcal{N}}{\mathcal{L}(A, C, S) + (\mathcal{N} - 1) \cdot \mathcal{R}_{L1,l}} \cdot \mathcal{C}_{size} \tag{9}$$

Where O_{size} is the operand size and $\mathcal{N} = \mathcal{C}_{size}/\mathcal{O}_{size}$ is the number of operands that can be fitted into a cache line. Eq. (10) is valid for Intel systems. On AMD L1 is write-through, in which case $\mathcal{R}_{L2,l}$ would replace $\mathcal{R}_{L1,l}$.

5 Performance Analysis

We now illustrate the results and provide several surprising insights into the performance characteristics of the tested atomics. Here, we exclude the results that show similar performance trends; For completeness we include all the relevant results in the Appendix. The latency results can be found in Figures 10-19. The bandwidth results can be found in Figures 20-29. 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.

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

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

5.1 Latency

First, we present a selection of the latency results. We compare CAS, FAD, and SWP that access cache lines in the E, M, and S coherency states; the O coherency state is also included for AMD. We exclude the F state from the Intel analysis as it only starts to affect the performance when more than two CPUs are used [19] while our tested systems 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, 3, and 4. The results indicate the correctness of the model predictions. We observe that atomics are consistently slower than reads by ≈5-10ns on both systems for the E and M states (cf. Figures 2a and 3a). From this we conjecture that atomics trigger a read for ownership and the latency difference between atomics and simple reads stems from $\mathcal{E}$, as predicted by both Eq. (1) and (2). 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, annihilating a need for invalidations. The difference in latency impacts the performance of local L1 cache accesses where the read latency is ≈1-2ns (see Figure 3a). It does not significantly influence accesses to remote caches or memory where latencies are well above 60ns. As predicted by out model and contrary to the common view, CAS has the same latency as FAD/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 FAD/SWP (a similar latency jump between L1 and L2 is visible for reads; see Figures 4a-4c). 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, 3a, 4a-4c. 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. (7)).

Cache lines in the M state are written back when evicted updating the core valid bits. Thus, there is no invalidation latency 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 and E states have ≈50ns higher latency than that of another core on the same CPU; see Figures 4a-4c. This is due to $\mathcal{H}$. For cache lines in the M state 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 in Figure 5. The results mostly match the model predictions. Atomics are again slower than reads in each case. However, the difference between the read latency and that of CAS/FAD is not the same for all the cache levels. CAS and FAD take ≈8ns longer than reads into the cache of a different core. Yet, for the local cache they consistently take ≈20ns longer than respective reads (cf. Figures 5a/5b and 5c). We attribute this surprising result to variable overheads related to locking a cache line.

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 module, a special 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 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 L1 or 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 Discussion & Insights

The latency evaluation illustrates novel insights. It turns out that, contrary to the common view [21], the latency of CAS, FAD, and SWP is in most cases identical and sometimes (L1 on Haswell and L3/memory on AMD) CAS is *faster* than FAD. This illustrates that atomics with different consensus numbers may still entail similar overheads. Additional overheads in CAS are due to fetching an additional argument from caches or memory.

The analysis also suggests several potential improvements for the hardware implementation of atomics. For example, we illustrate 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 issued CASes, thus requiring the invalidations. Another potential strategy would not utilize invalidations before executing CASes. As unsuccessful CASes usually constitute a crucial part of all the issued CASes in various parallel designs [21], this might accelerate some workloads.

5.2 Bandwidth

We now analyze a selection of bandwidth results. Due to space constraints we illustrate the Haswell results for the M state (see Figure 6) and only briefly discuss Ivy Bridge and Bulldozer. Here, we compare atomics to writes.

Similarly to latency and again surprisingly, the bandwidth results for Haswell indicate that CAS is comparable or faster than FAD (≈0.04 GB/s). Moreover, the bandwidth of atomics is higher in higher level caches (for 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.

On both Intel architectures and the AMD system the bandwidth of atomics is significantly (≈5-30 times) lower than that of writes because the latter utilize ILP. However, in our benchmark design (see § 3.2) we specifically enabled the possibility of parallel execution of FAD/SWP. We conjecture that the hardware implementation of atomics prevents such parallelism, limiting performance.

5.2.1 Discussion & Insights

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

Another reason for the low bandwidths is that the caches are organized in cache lines of 64 byte size on all the tested systems. At most one access to each cache line cannot be executed in L1 in our bandwidth benchmarks. Therefore the bandwidth to L1 is an important performance factor. On all the tested systems the bandwidth and the latency of atomics to the L1 cache are however significantly lower than these of writes.

Finally, our results indicate that atomics do not allow for ILP whatsoever. Relaxing this restriction in some cases (e.g., for the independent executions of FAD 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. Both 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 that operates on 64bit operands would thus be desirable in the latency-constrained applications running on AMD Bulldozer.

5.4 Contention

We now evaluate the effect of many threads accessing the same cache line with atomics; see Figure 8. This benchmark targets the codes with highly contended shared counters and synchronization variables. Here, a selected number of threads issues atomics targeted at the same cache line.

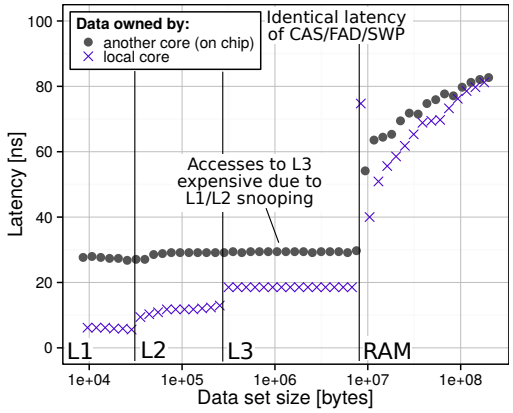
Intel Haswell reaches the accumulated contended bandwidth of over 100GB/s with four cores, Ivy Bridge has almost 80GB/s with three cores on two CPUs. These numbers are very close to the accumulated non-contended bandwidth. We conjecture that both Intel architectures detect that issued operations access the same cache line in an arbitrary order, annihilating the need for the actual execution of all the writes.

Bulldozer on the other hand suffers from the contention significantly when writing to the same cache line concurrently. The accumulated contended bandwidth of three cores on two CPUs is between 500 and 600 MB/s for writes. That bandwidth is lower than that of a single core writing to L1 and five times higher than that of FAD or CAS.

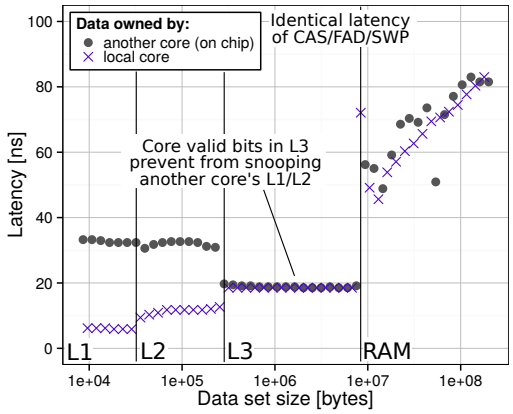
We conclude that all the considered architectures have significantly lower bandwidth in a contended execution of atomic operations than in a non-contended case. This may constitute a performance limitation in state-of-the-art multi- and manycore designs with massive thread-level parallelism.

5.5 Prefetchers and Other Mechanisms

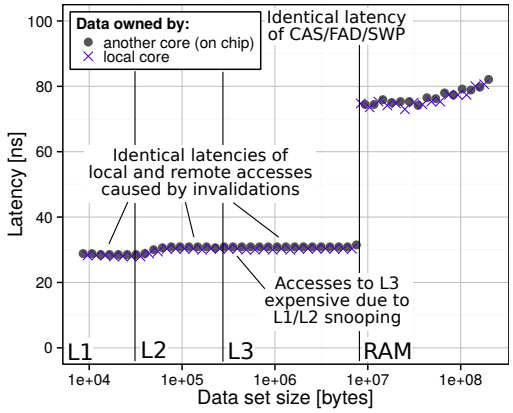
State-of-the-art architectures host different mechanisms that impact the CPU performance. For example, Intel Haswell deploys Hardware Prefetcher (prefetching data/instructions after successive L3 misses or after detecting cache hit patterns), Adjacent Cache Line Prefetcher (unconditional prefetching 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 atomic operations. We select Intel Haswell as the testbed and we skip the latency results because they are only marginally (≈1% of 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.



(a)SWP/CAS/FAD, Exclusive state

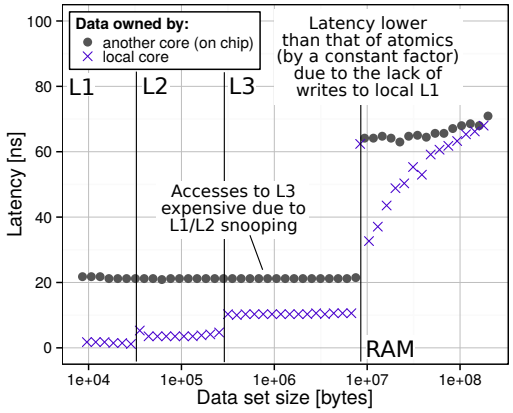


(b) SWP/CAS/FAD, Modified state

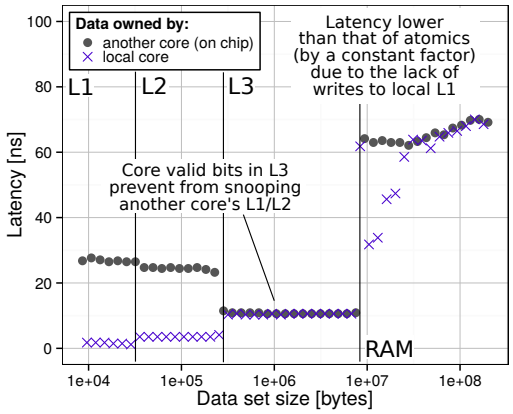


(c) SWP/CAS/FAD, Shared state

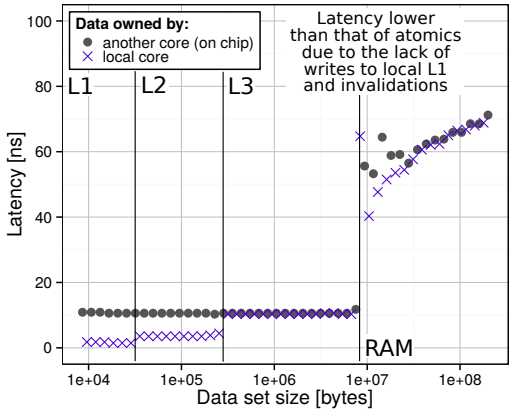
Figure 2: The comparison of the latency of CAS, FAD, SWP, and read on Haswell. The requesting core accesses its own cache lines (local) and cache lines of a different core from the same chip (on chip).



(a) read, Exclusive state

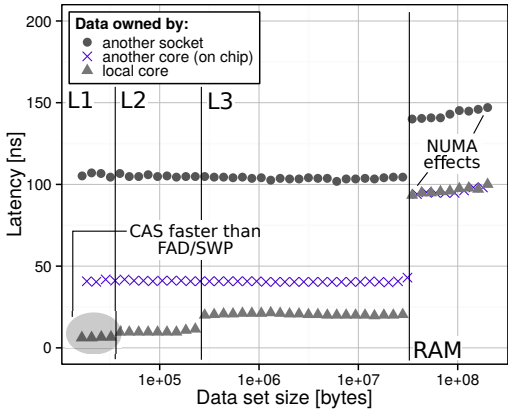


(b) read, Modified state

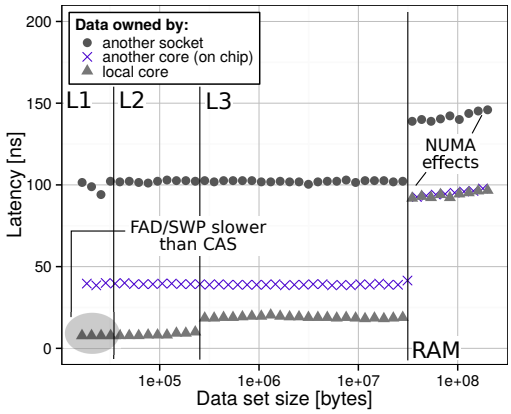


(c) read, Shared state

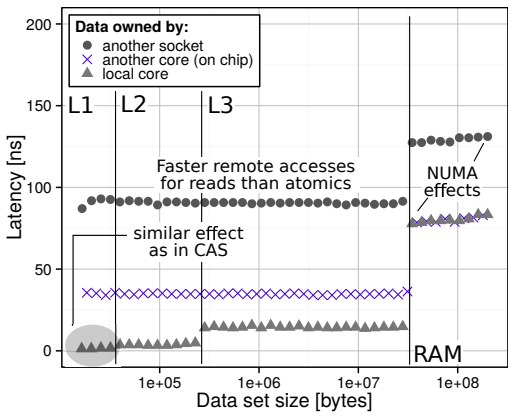
Figure 3: The comparison of the latency of CAS, FAD, SWP, and read on Haswell. The requesting core accesses its own cache lines (local) and cache lines of a different core from the same chip (on chip).



(a) CAS, Exclusive state

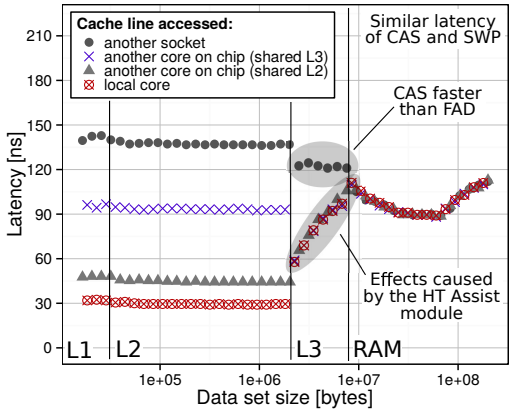


(b) SWP/FAD, Exclusive state

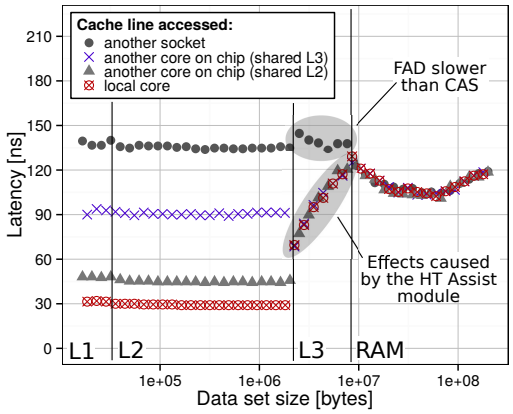


(c) read, Exclusive state

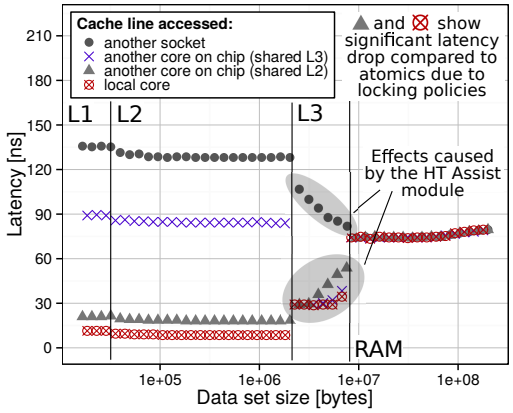
Figure 4: The comparison of the latency of CAS, FAD, SWP, and read on Ivy Bridge. The requesting core accesses its own cache lines (local), cache lines of a different core from the same chip (on chip), and cache lines of a different core from a different socket (other socket).



(a) CAS/SWP, Exclusive state

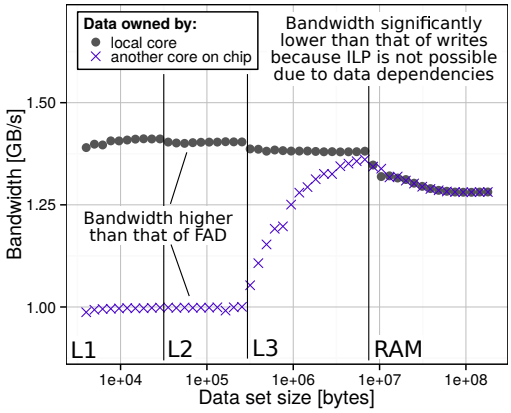


(b) FAD, Exclusive state

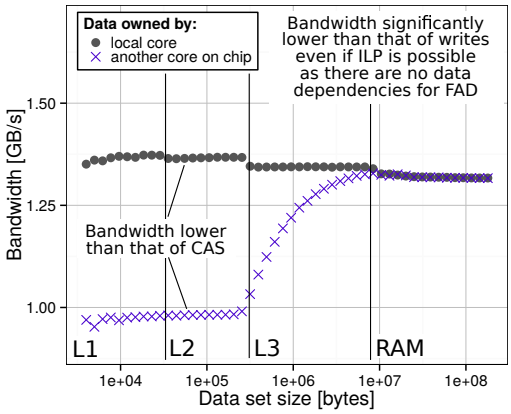


(c) read, Exclusive state

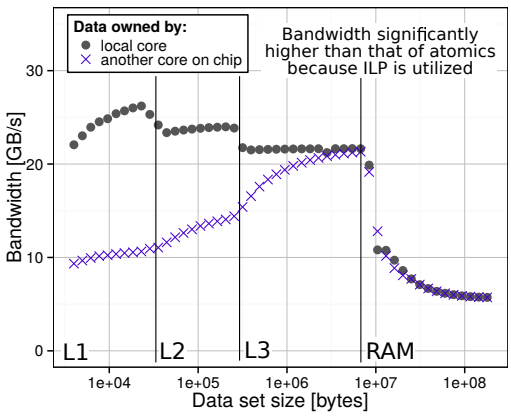
Figure 5: The comparison of the latency of CAS, FAD, SWP, and read on Bulldozer. The requesting core accesses its own cache lines (local), cache lines of different cores that share L2 and L3 with the requesting core (on chip, shared L2 and L3, respectively), and cache lines of a different core from a different socket (other socket).



(a) CAS, Modified state

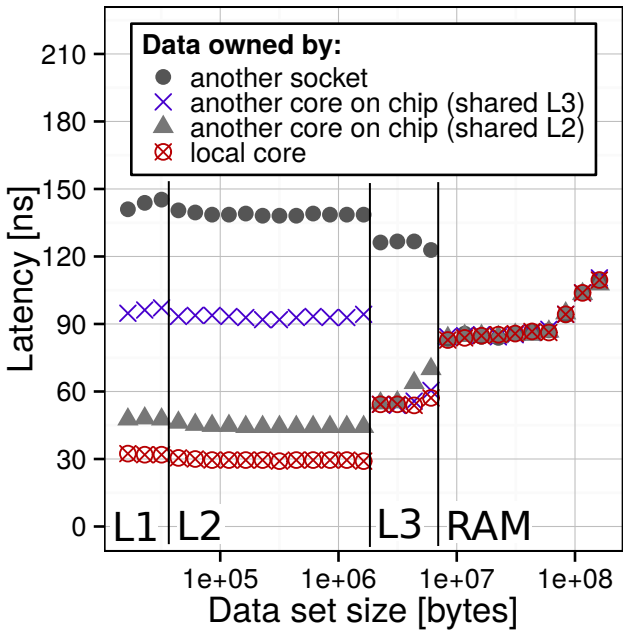


(b) FAD, Modified state

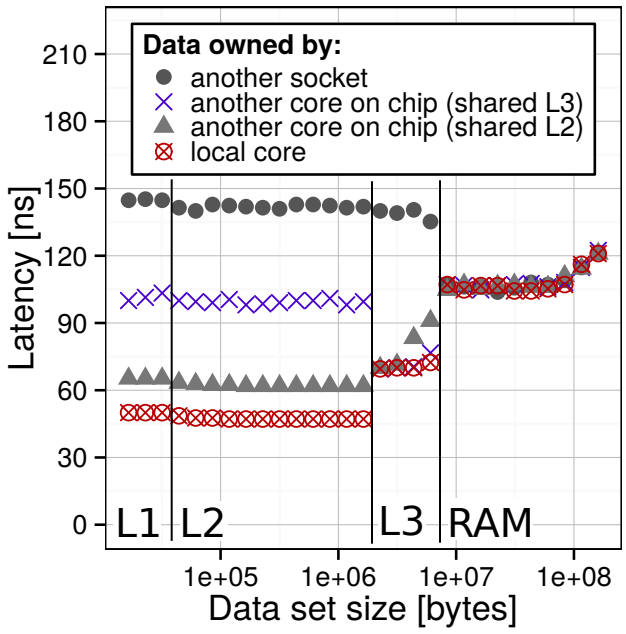


(c) read, Modified state

Figure 6: The comparison of the bandwidth of CAS, FAD, 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).

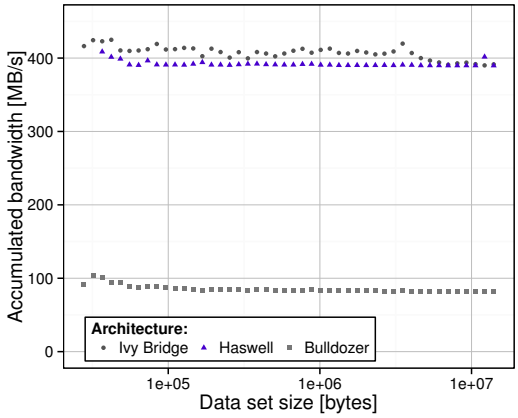


(a) CAS (64 bits)

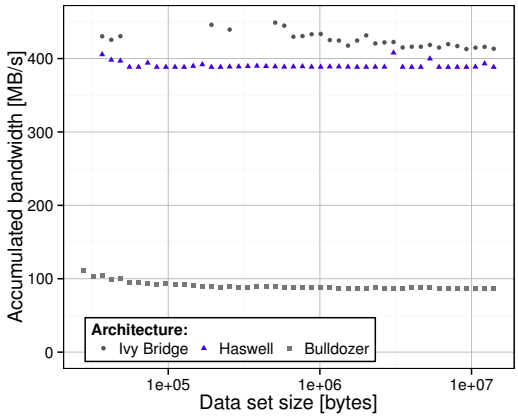


(b) CAS (128 bits)

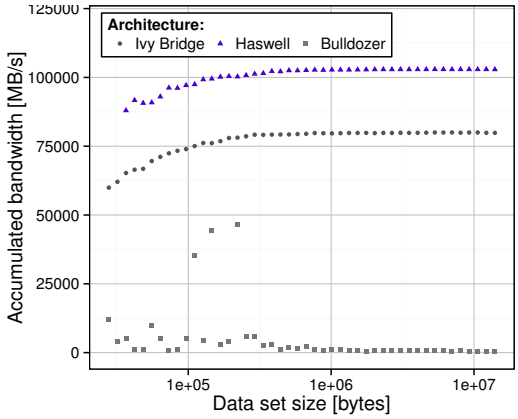
Figure 7: The comparison of the latency of CAS using operands of 64 and 128 bits in size (AMD Bulldozer, the M state).



(a) CAS.

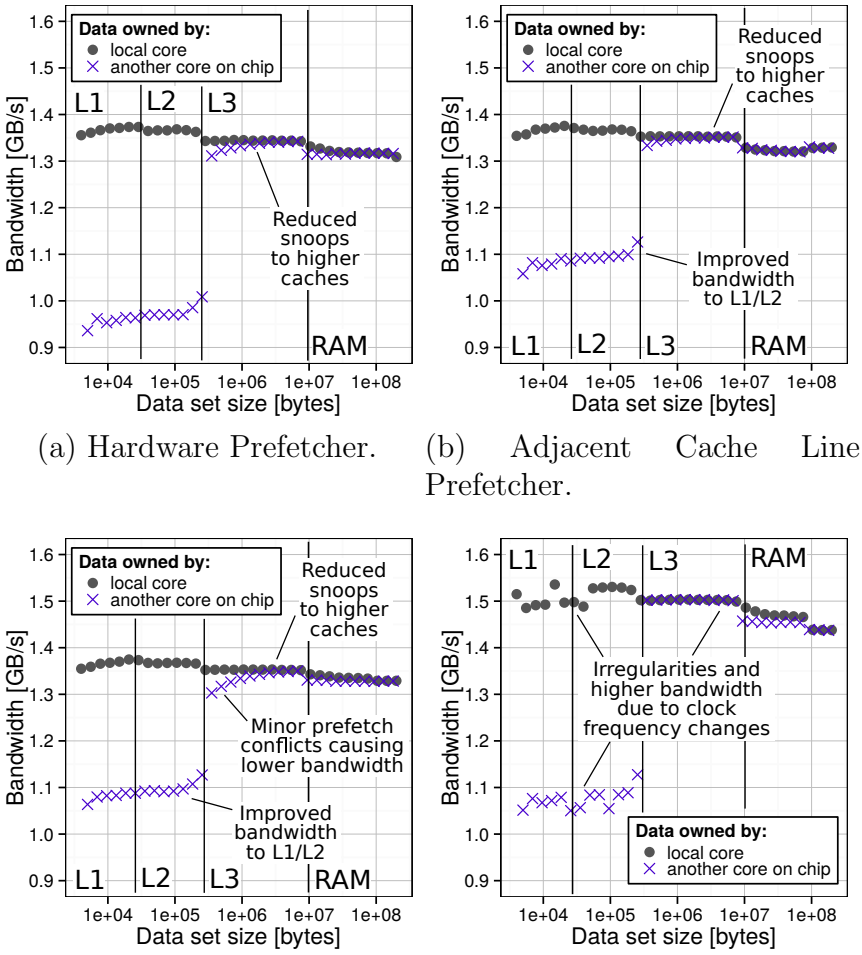


(b) FAD



(c) write

Figure 8: The comparison of contended bandwidth of CAS, FAD, and writes on all the tested architectures.



(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 FAD (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.

6 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. [21]. This work uses the comparison to motivate the proposed parallel queue that extensively utilizes Fetch-And-Add. It differs from the study in this work as it only illustrates the lower performance of CAS caused by the semantics that introduce wasted work.

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. [20]. A comparison of the performance of memory accesses on Intel Nehalem and AMD Shanghai was performed by Molka et al. [9]; a similar study targets the AMD Bulldozer and Intel Sandy Bridge microarchitectures [19]. Other analyses on the performance of the memory subsystems include the work by Babka et al. [2], Pend et al. [23], and Hristea et al. [12]. 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. [21], a hierarchical lock by Luchangco et al. [17], and a queue by Michael and Scott [18]. Many fundamental structures and designs can be found in a book by Herlihy and Shavit [11].

Finally, the considered architectures and cache coherency protocols are extensively described in various manuals and papers [1, 8, 13–15, 25]. Several performance models targeting on-chip communication have been introduced, for example a model by Garea and Hoefler [6]. The model proposed in this work differs from that work because it specifically targets latency and bandwidth of atomic operations in the onnode environment.

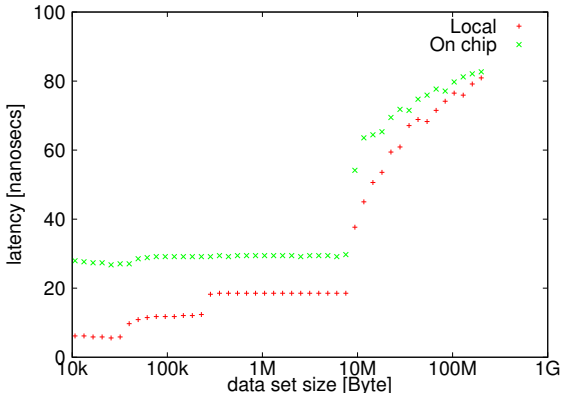
7 Conclusion

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 manycore environments.

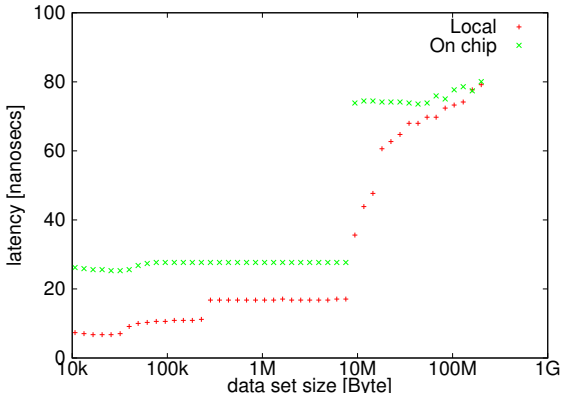
In this work we propose a performance model and provide a detailed evaluation of the latency and bandwidth of several atomic operations (Compare-And-Swap, Fetch-And-Add, Swap) that validates the model. The selected atomics are widely utilized in various parallel codes such as graph traversals, shared counters, spinlocks, and numerous data structures. Our performance insights include the observation that CAS and FAD 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 all the atomics flush write buffers and prevent from instruction level parallelism, significantly limiting the bandwidth (up to 20x in comparison with simple writes). Our analysis can thus be used for designing more performant parallel systems.

The results also indicate several potential improvements in the design of the caching hierarchy. For example, the AMD Bulldozer architecture limits its performance with useless 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. We believe our analysis and data can be used by architects and engineers to develop more performant memory subsystems that would offer even higher speedups for parallel workloads.

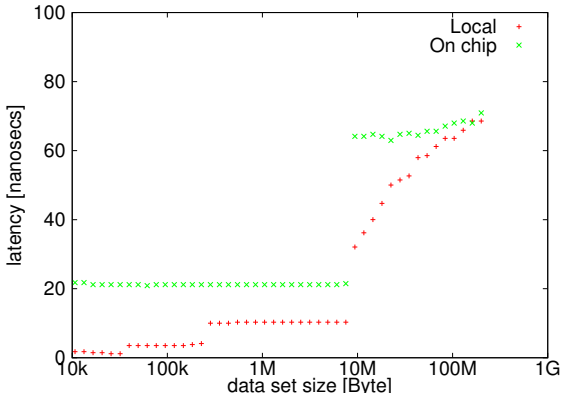
8 Appendix



(a)CAS, Exclusive state

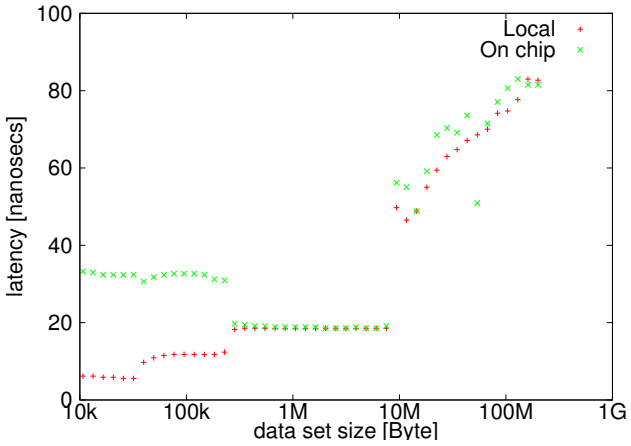


(b) FAD/SWP, Exclusive state

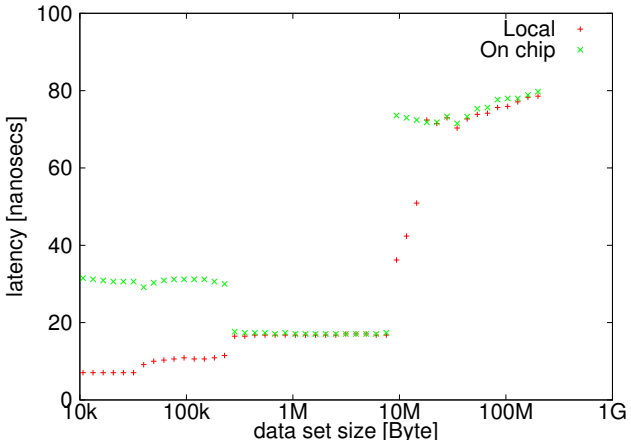


(c) read, Exclusive state

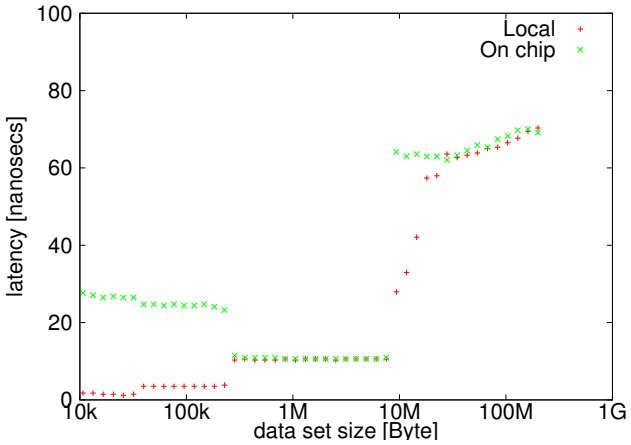
Figure 10: Latency of CAS, FAD, SWP, and read on Haswell



(a) CAS, Modified state

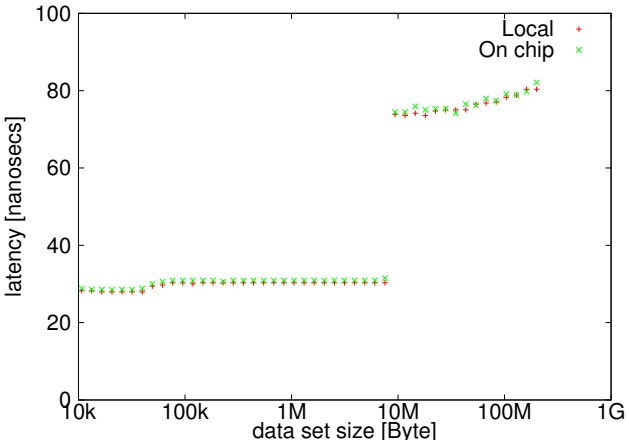


(b) FAD/SWP, Shared state

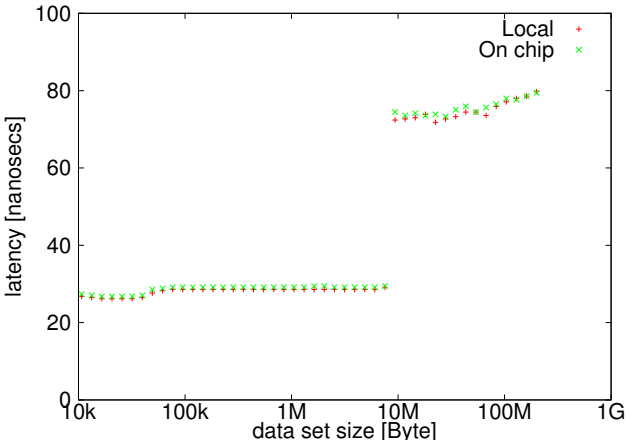


(c) read, Modified state

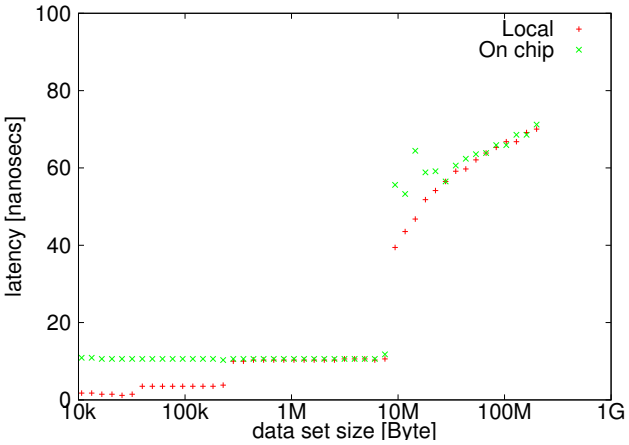
Figure 11: Latency of CAS, FAD, SWP, and read on Haswell



(a) CAS, Shared state

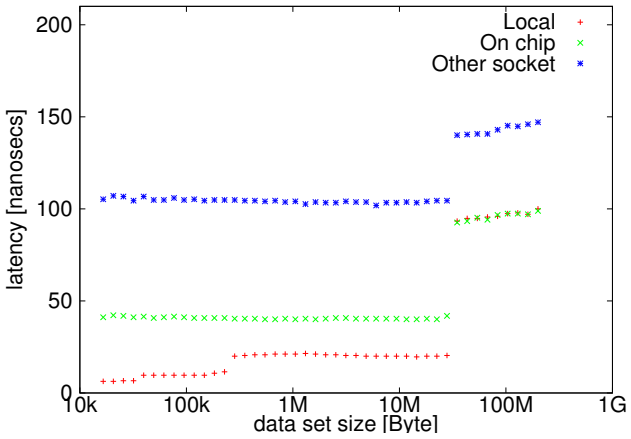


(b) FAD/SWP, Shared state

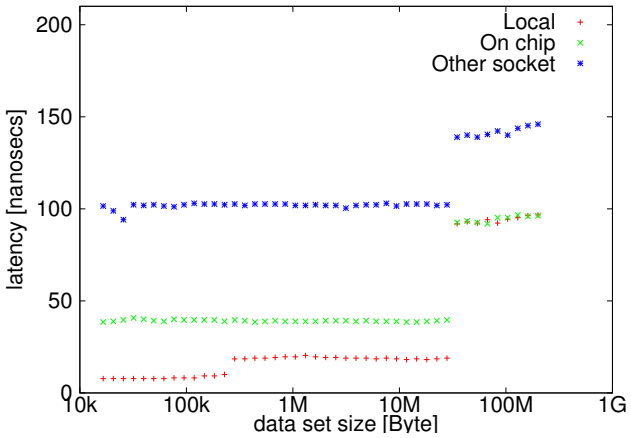


(c) read, Shared state

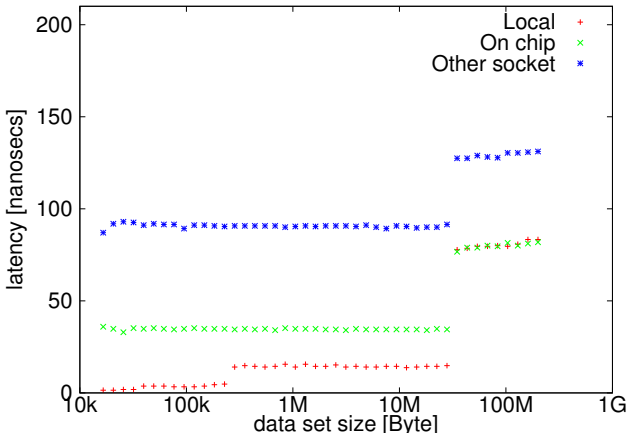
Figure 12: Latency of CAS, FAD, SWP, and read on Haswell



(a)CAS, Exclusive state

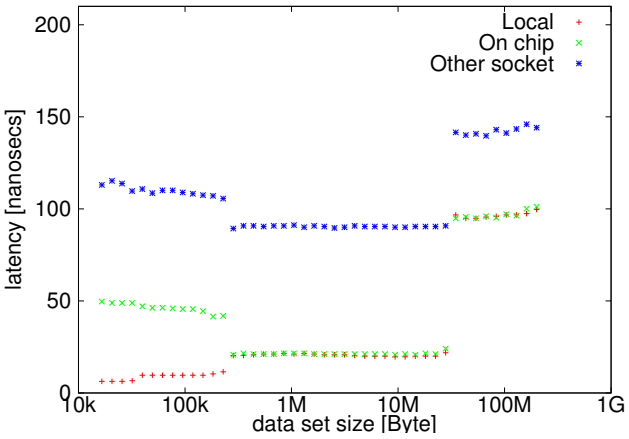


(b) FAD/SWP, Exclusive state

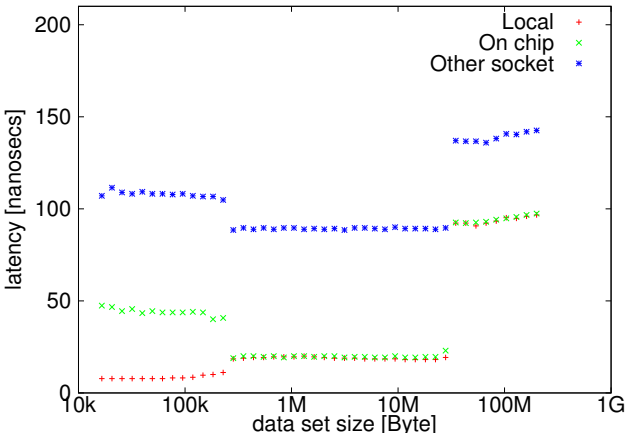


(c) read, Exclusive state

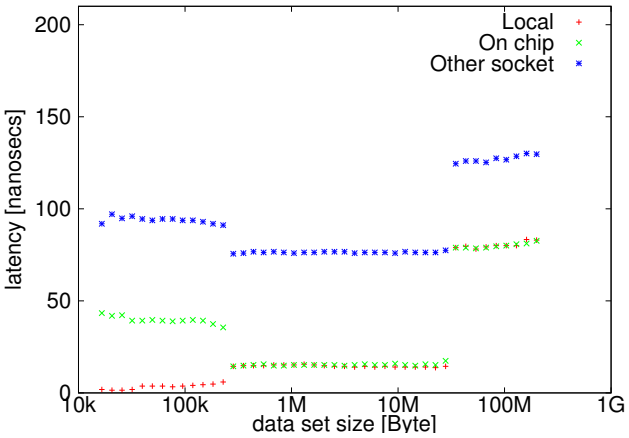
Figure 13: Latency of CAS, FAD, SWP, and read on Ivy bridge



(a) CAS, Modified state

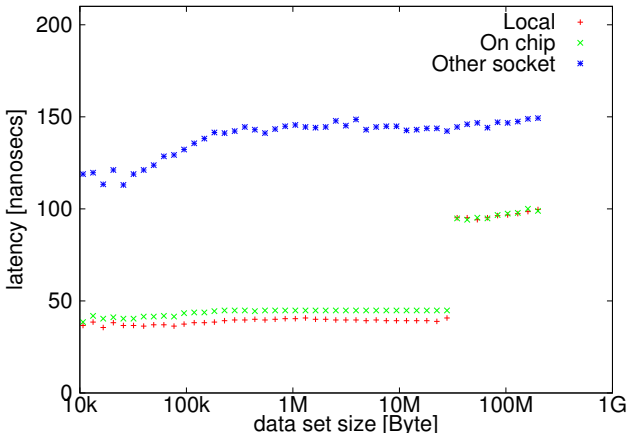


(b) FAD/SWP, Shared state

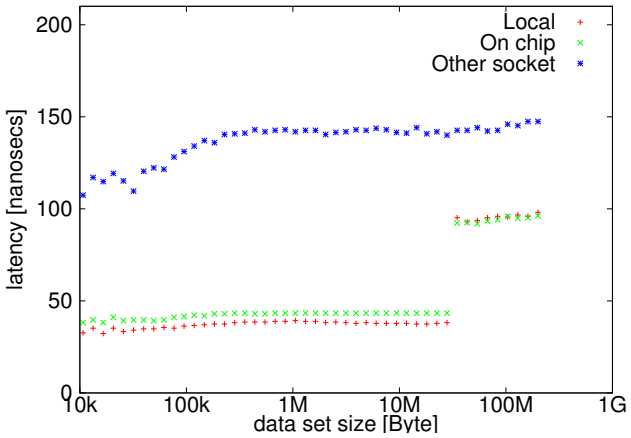


(c) read, Modified state

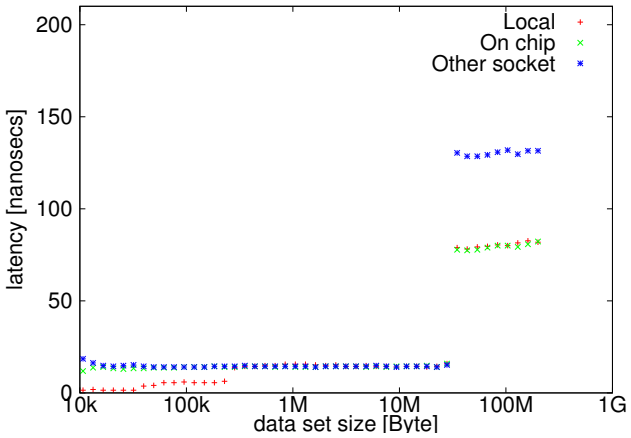
Figure 14: Latency of CAS, FAD, SWP, and read on Ivy bridge



(a) CAS, Shared state

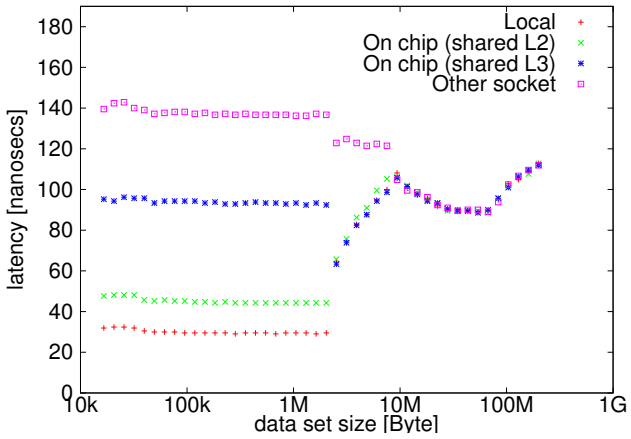


(b) FAD/SWP, Shared state

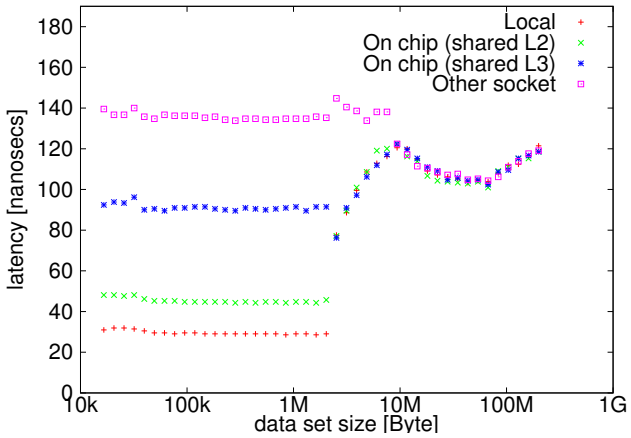


(c) read, Shared state

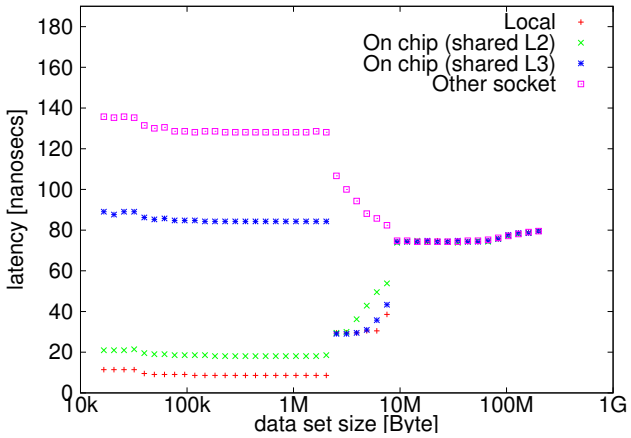
Figure 15: Latency of CAS, FAD, SWP, and read on Ivy bridge



(a) CAS, Exclusive state

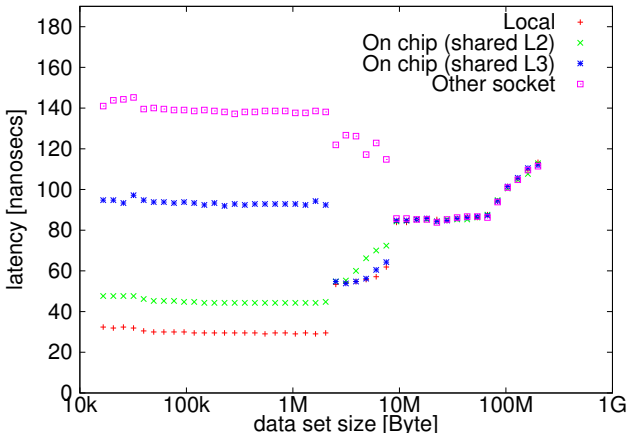


(b) FAD/SPW, Exclusive state

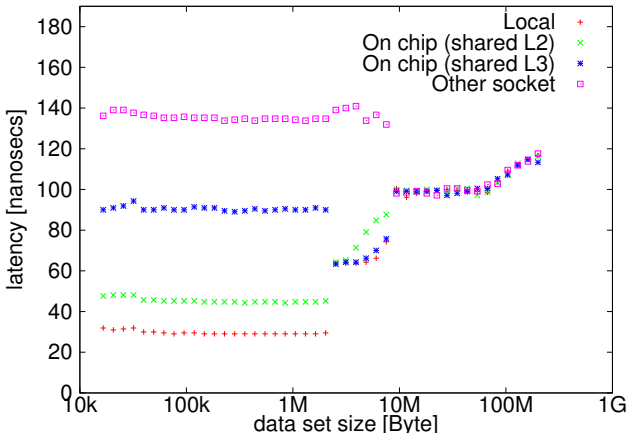


(c) read, Exclusive state

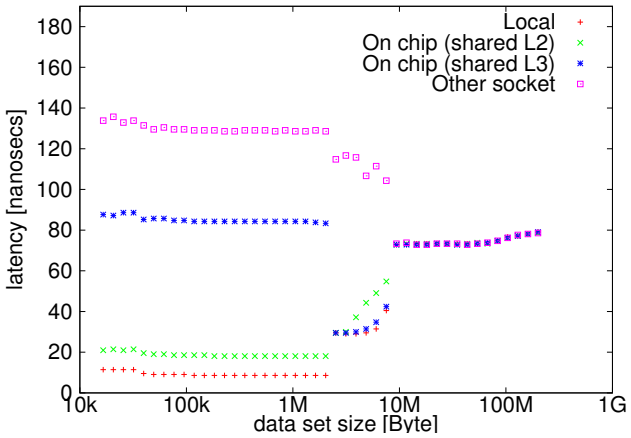
Figure 16: Latency of CAS, FAD/SPW, SWP, and read on AMD Bulldozer



(a) CAS, Modified state

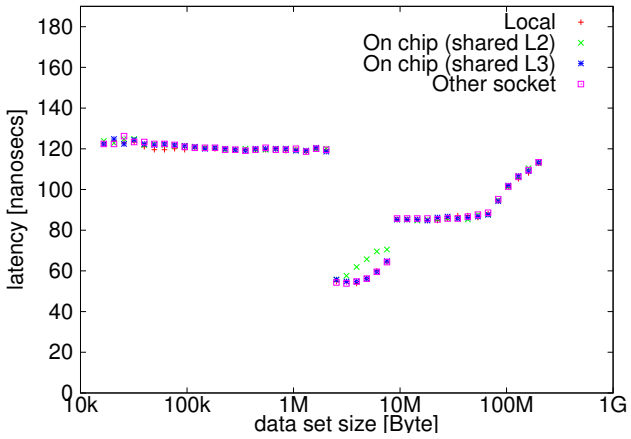


(b) FAD/SPW, Modified state

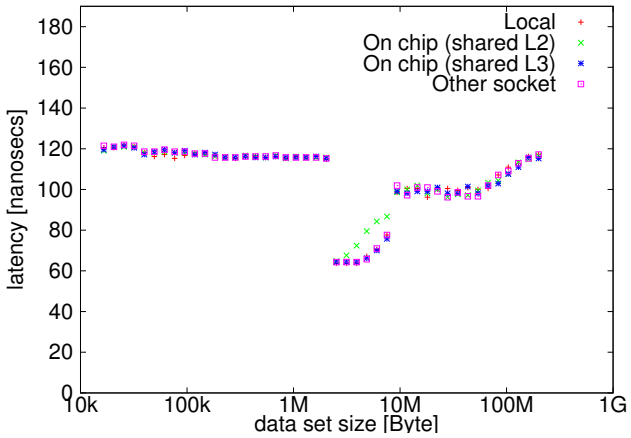


(c) read, Modified state

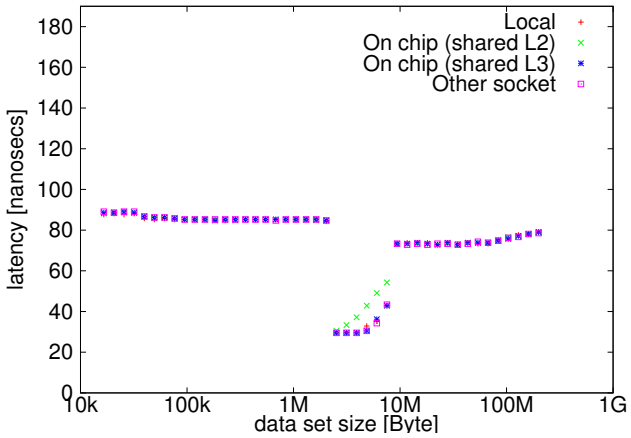
Figure 17: Latency of CAS, FAD/SPW, SWP, and read on AMD Bulldozer



(a) CAS, Owned state

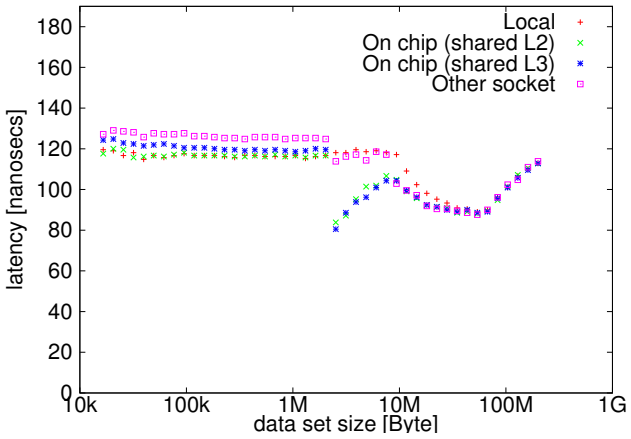


(b) FAD/SPW, Owned state

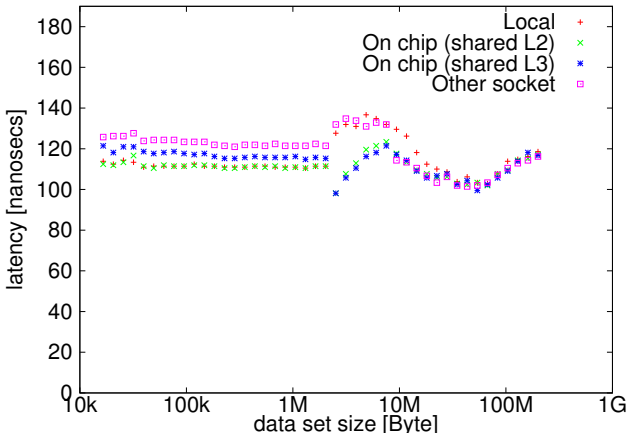


(c) read, Owned state

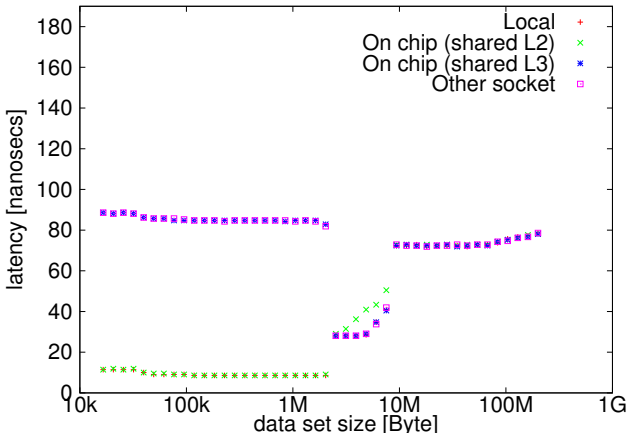
Figure 18: Latency of CAS, FAD/SPW, SWP, and read on AMD Bulldozer



(a) CAS, Shared state

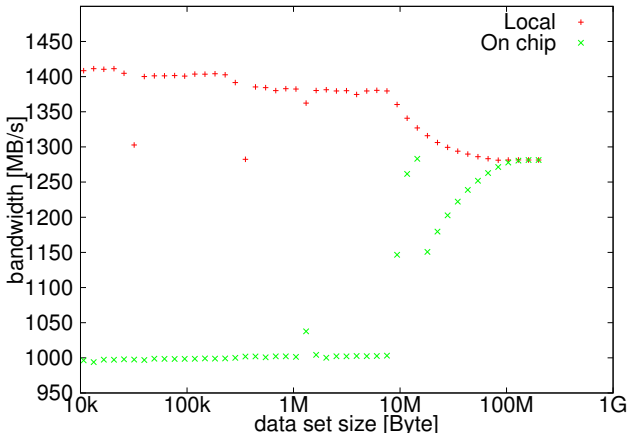


(b) FAD/SPW, Shared state

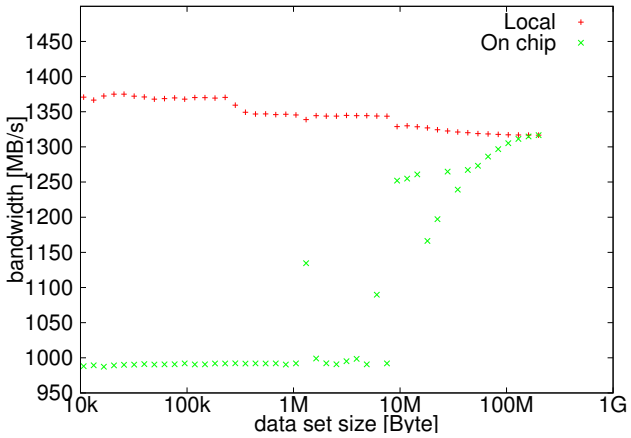


(c) read, Shared state

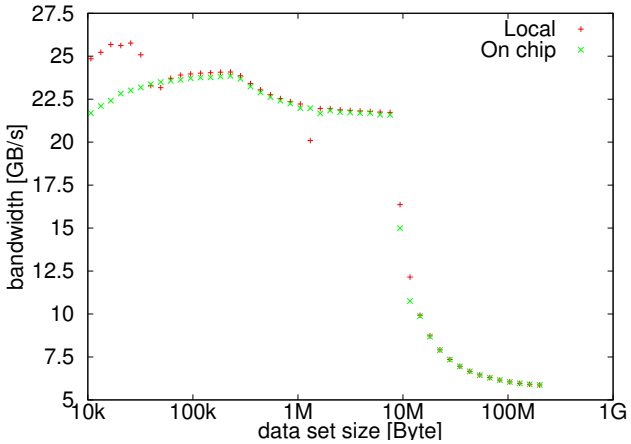
Figure 19: Latency of CAS, FAD/SPW, SWP, and read on AMD Bulldozer



(a) CAS, Exclusive

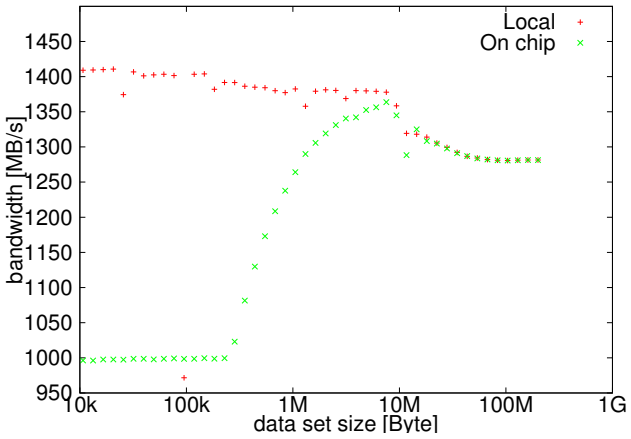


(b) FAD, Exclusive

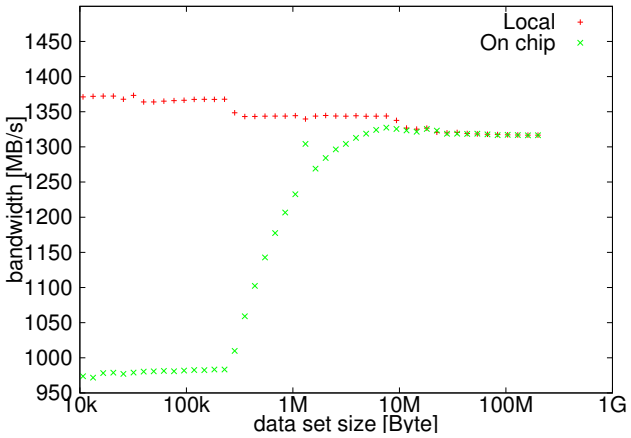


(c) write, Exclusive

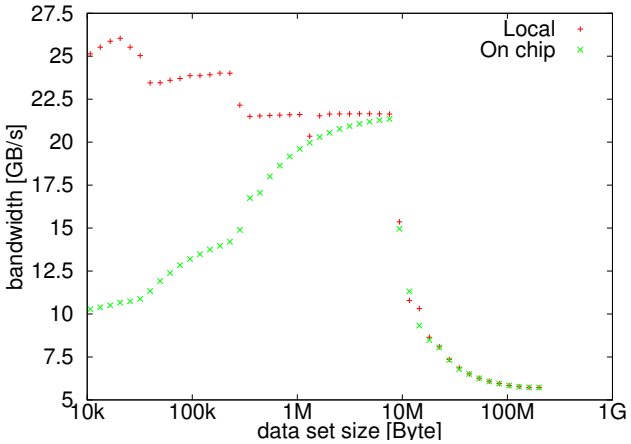
Figure 20: Bandwidth of Compare-and-Swap, Fetch-and-Add, and writes on Haswell



(a) CAS, Modified

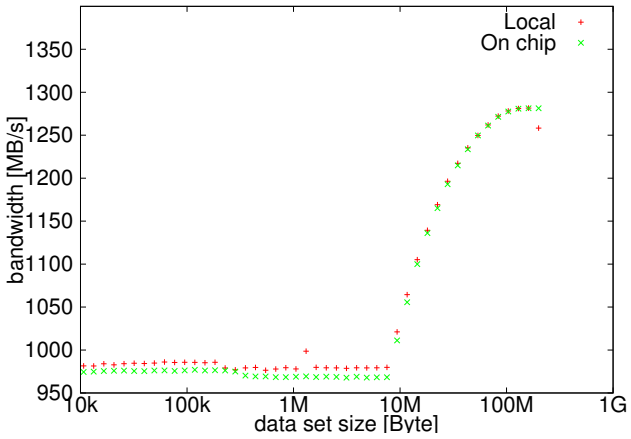


(b) FAD, Modified

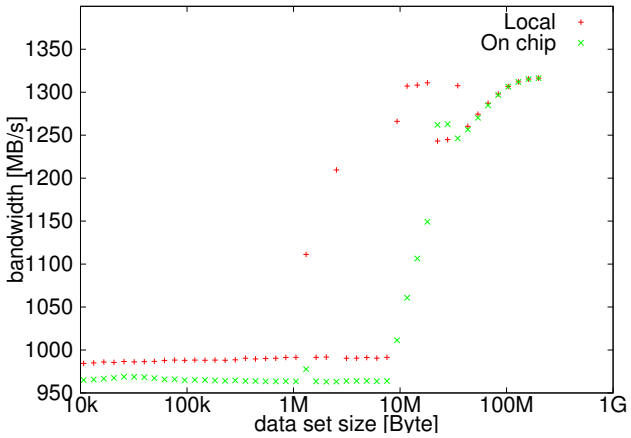


(c) write, Modified

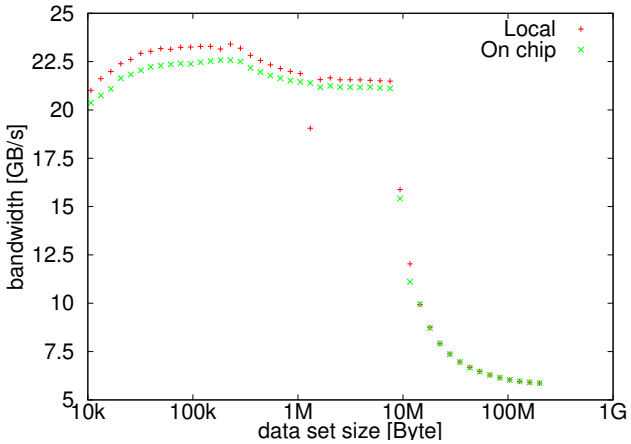
Figure 21: Bandwidth of Compare-and-Swap, Fetch-and-Add, and writes on Haswell



(a) CAS, Shared

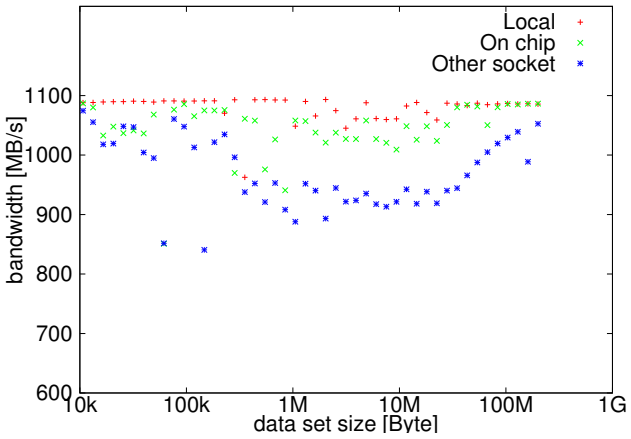


(b) FAD, Shared

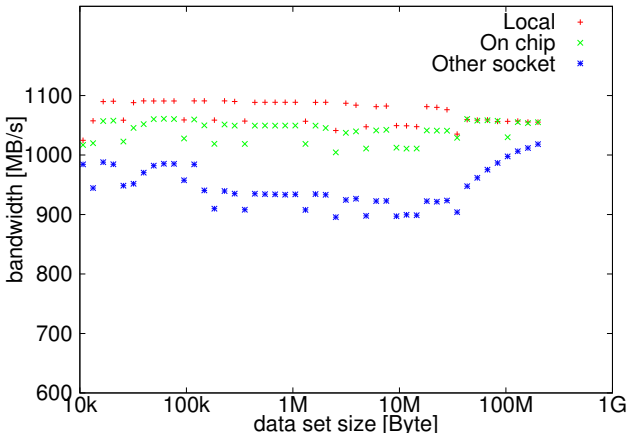


(c) write, Shared

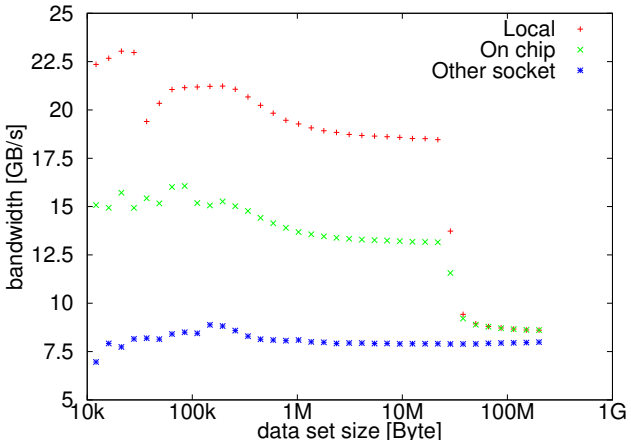
Figure 22: Bandwidth of Compare-and-Swap, Fetch-and-Add, and writes on Haswell



(a) CAS, Exclusive

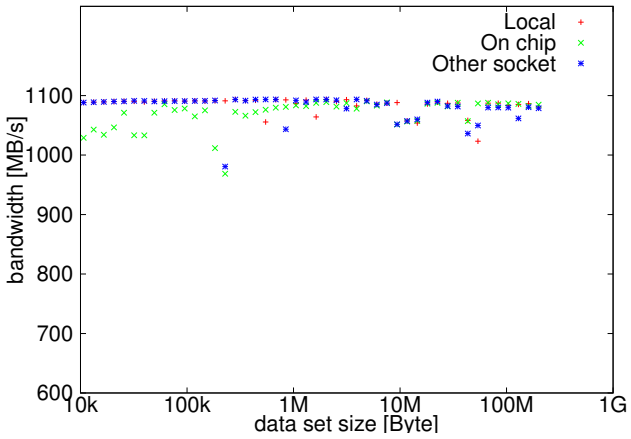


(b) FAD, Exclusive

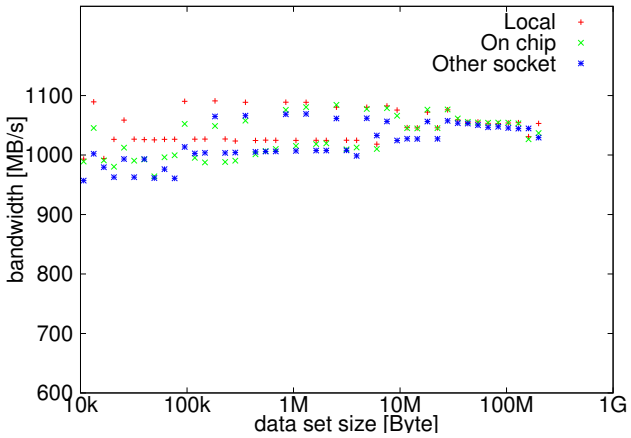


(c) write, Exclusive

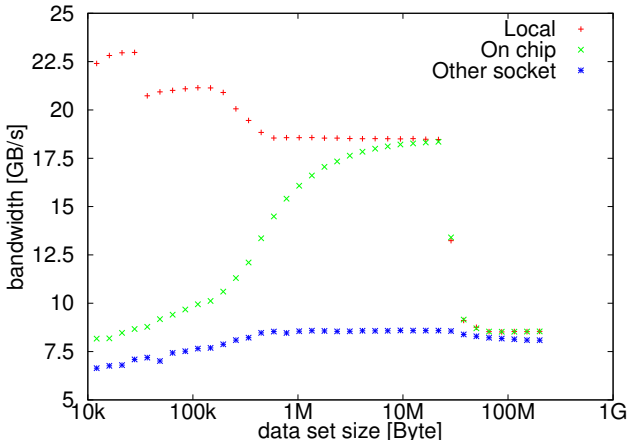
Figure 23: Bandwidth of Compare-and-Swap, Fetch-and-Add, and writes on Ivy bridge



(a) CAS, Modified

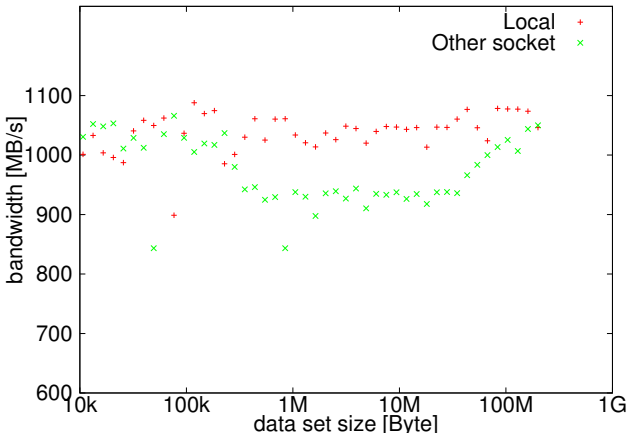


(b) FAD, Modified

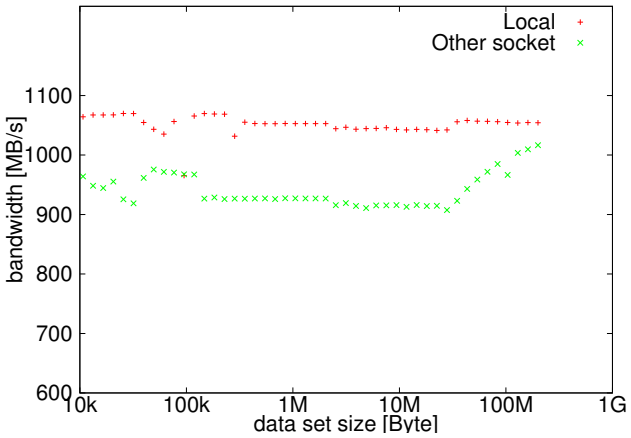


(c) write, Modified

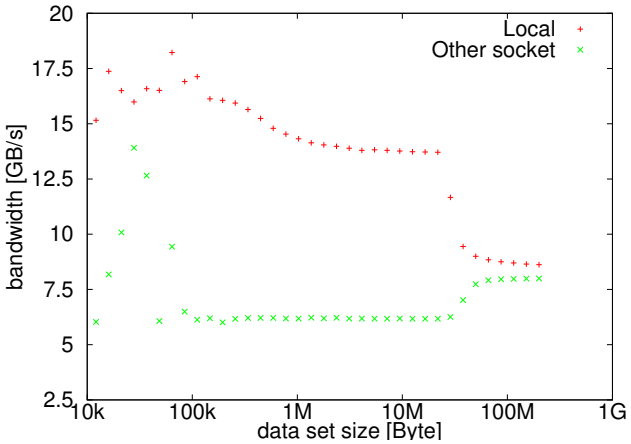
Figure 24: Bandwidth of Compare-and-Swap, Fetch-and-Add, and writes on Ivy bridge



(a) CAS, Shared

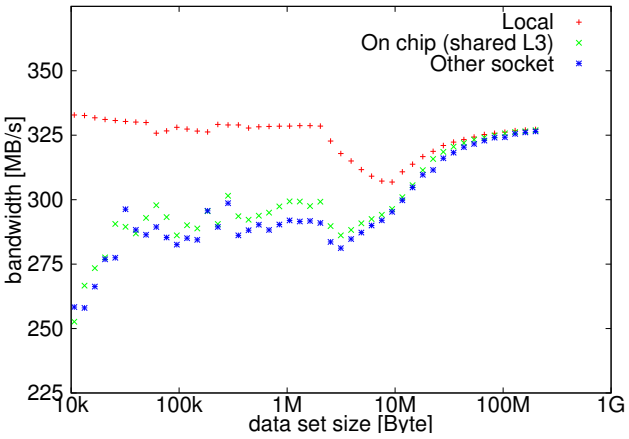


(b) FAD, Shared

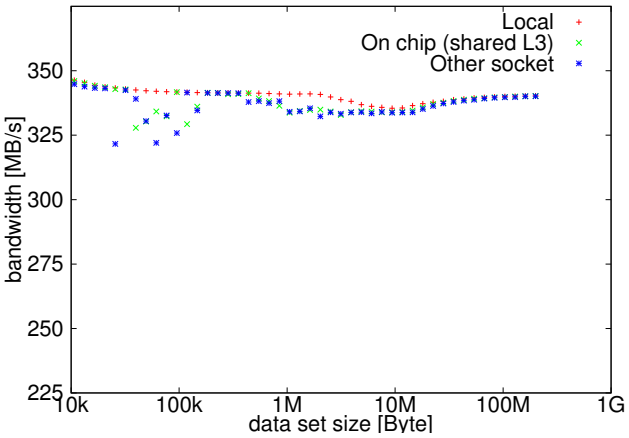


(c) write, Shared

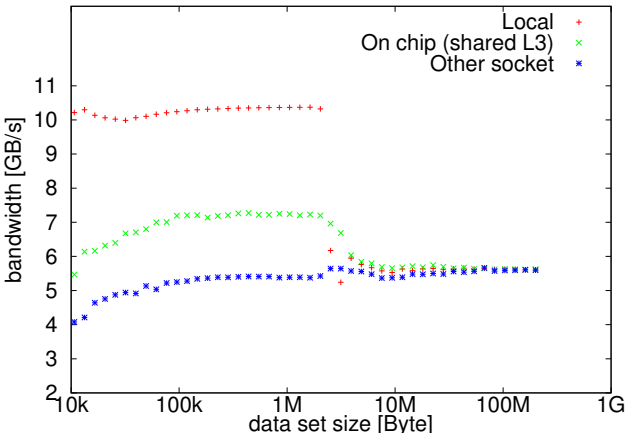
Figure 25: Bandwidth of Compare-and-Swap, Fetch-and-Add, and writes on Ivy bridge



(a) CAS, Exclusive

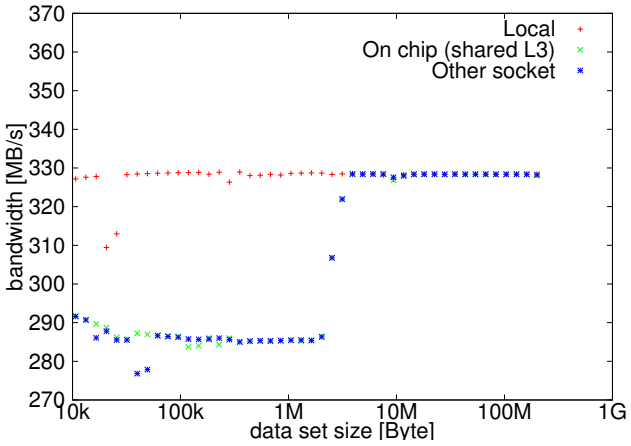


(b) FAD/SWP, Exclusive

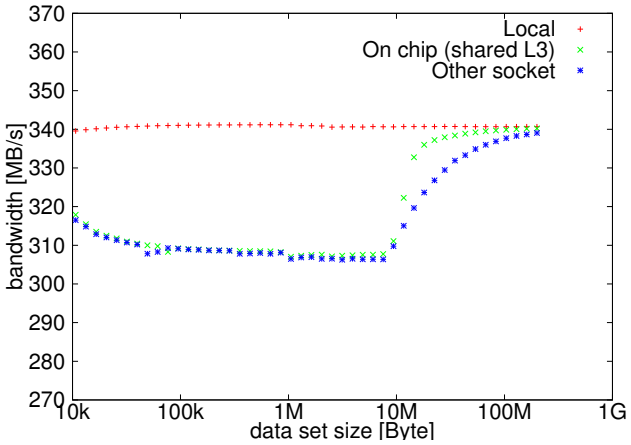


(c) write, Exclusive

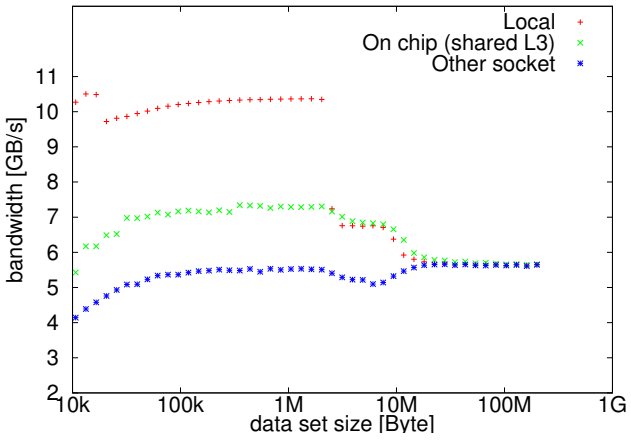
Figure 26: Bandwidth of Compare-and-Swap, Fetch-and-Add, Swap, and writes on Bulldozer



(a) CAS, Modified

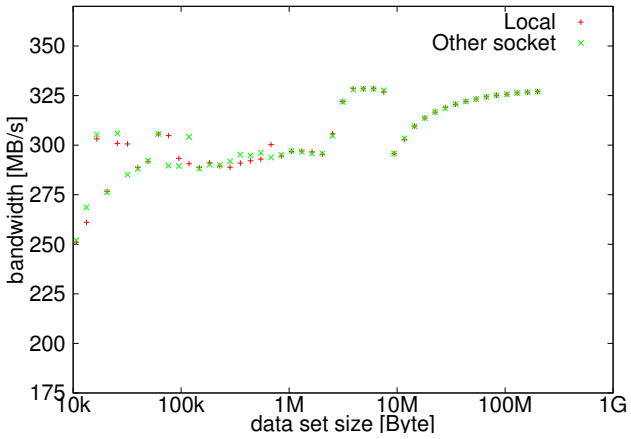


(b) FAD/SWP, Modified

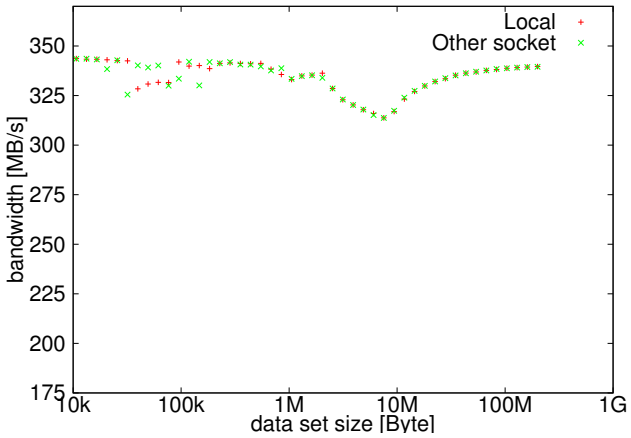


(c) write, Modified

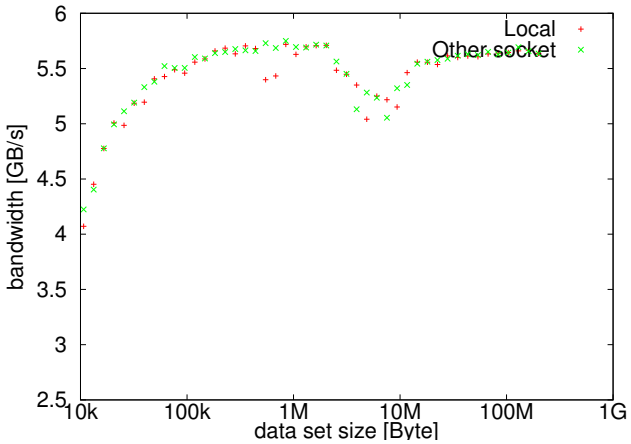
Figure 27: Bandwidth of Compare-and-Swap, Fetch-and-Add, Swap, and writes on Bulldozer



(a) CAS, Owned

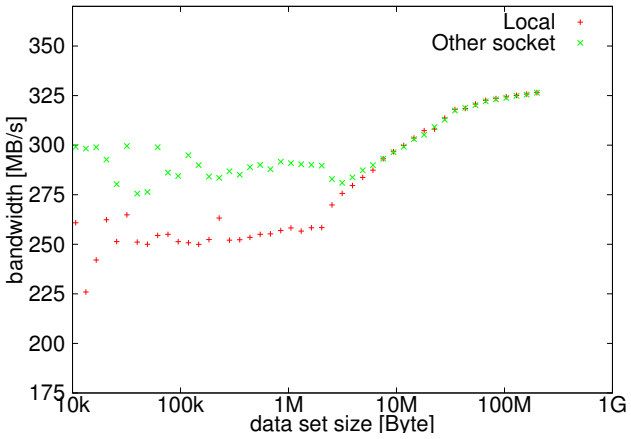


(b) FAD/SWP, Owned

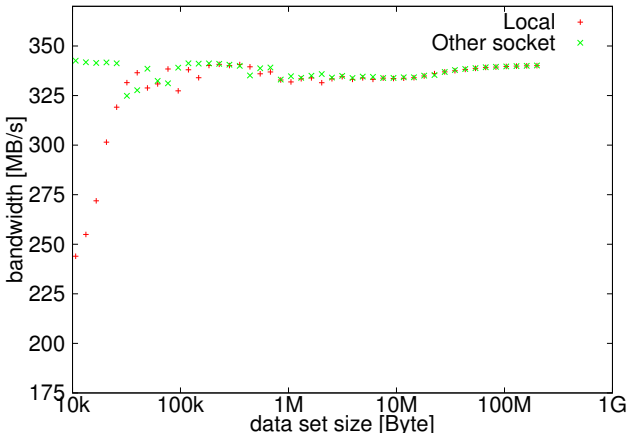


(c) write, Owned

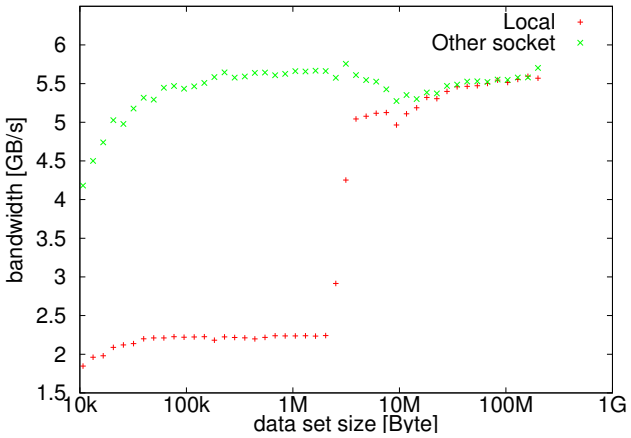
Figure 28: Bandwidth of Compare-and-Swap, Fetch-and-Add, Swap, and writes on Bulldozer



(a) CAS, Shared



(b) FAD/SWP, Shared



(c) write, Shared

Figure 29: Bandwidth of Compare-and-Swap, Fetch-and-Add, Swap, and writes on Bulldozer

References

[1] AMD Corporation. Software Optimization Guide for AMD Family 15h Processors, January 2014. available at: http://support.amd.com/TechDocs/47414_15h_sw_opt_guide.pdf.

[2] V. Babka and P. Tůma. Investigating cache parameters of x86 family processors. In *Proceedings of the 2009 SPEC Benchmark Workshop on Computer Performance Evaluation and Benchmarking*, pages 77–96, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[4] I. Corporation. Intel Xeon E5-2600 v3 Codename Haswell-EP Launch, 2014. Intel White Paper.

[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] S. R. Garea and T. Hoefler. Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi. Jun. 2013. Accepted at HPDC'13.

[7] 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.

[8] J. R. Goodman and H. H. Hum. Forward state for use in cache coherency in a multiprocessor system, July 26 2005. US Patent 6,922,756.

[9] D. Hackenberg, D. Molka, and W. E. Nagel. Comparing Cache Architectures and Coherency Protocols on x86-64 Multicore SMP Systems. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, pages 413–422, New York, NY, USA, 2009. ACM.

[10] T. L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing*, DISC '01, pages 300–314, London, UK, UK, 2001. Springer-Verlag.

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

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

[13] Intel Corporation. Intel 64 and IA-32 Architectures Optimization Reference Manual, September 2014. available at: http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[14] Intel Corporation. Intel 64 and IA-32 Architectures Software Developer's Manual, September 2014. available at: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html.

[15] T. Jain and T. Agrawal. The haswell microarchitecture–4th generation processor. *International Journal of Computer Science and Information Technologies*, 4(3):477–480, 2013.

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

[17] V. Luchangco, D. Nussbaum, and N. Shavit. A Hierarchical CLH Queue Lock. In *Proceedings of the 12th International Conference on Parallel Processing*, Euro-Par'06, pages 801–810, Berlin, Heidelberg, 2006. Springer-Verlag.

[18] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[19] D. Molka, D. Hackenberg, and R. Schöne. Main Memory and Cache Performance of Intel Sandy Bridge and AMD Bulldozer. In *Proceedings of the Workshop on Memory Systems Performance and Correctness*, MSPC '14, pages 4:1–4:10, New York, NY, USA, 2014. ACM.

[20] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 261–270, Washington, DC, USA, 2009. IEEE Computer Society.

[21] A. Morrison and Y. Afek. Fast Concurrent Queues for x86 Processors. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 103–112, New York, NY, USA, 2013. ACM.

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

[23] 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.

[24] D. Slogsnat, A. Giese, and U. Brüning. A Versatile, Low Latency HyperTransport Core. In *Proceedings of the 2007 ACM/SIGDA 15th International Symposium on Field Programmable Gate Arrays*, FPGA '07, pages 45–52, New York, NY, USA, 2007. ACM.

[25] T. Suh, D. M. Blough, and H.-H. Lee. Supporting cache coherence in heterogeneous multiprocessor systems. In *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, volume 2, pages 1150–1155. IEEE, 2004.

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

[27] R. Wolf. Nasa Pleiades Infiniband Communications Network, 2009. Intl. ACM Symposium on High Performance Distributed Computing.

[28] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance Evaluation of Intel Transactional Synchronization Extensions for High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 19:1–19:11, New York, NY, USA, 2013. ACM.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Modelling and Evaluating Performance of Atomic Operations

Authored by (in block letters):

For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Schweizer	Hermann

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Zollbrugg, 11.01.2015	H. Schweizer

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.