# A LOCK-FREE CONCURRENT PRIORITY QUEUE

*Renato Marroquín, Aristeidis Mastoras, David Sidler*

Department of Computer Science
ETH Zürich
Zürich, Switzerland

## ABSTRACT

A priority queue appears in various multi-threading applications. However, an efficient implementation for a concurrent priority queue is challenging in practice, since already proposed mutual exclusion strategies cause blocking and have bad scalability.

We present a lock-free implementation of a concurrent priority queue which only makes use of atomic operations which are present in most modern x86 CPUs. The underlying data structure is a skip list, enabling lookups in $\mathcal{O}(\log n)$ and *pop* operations in $\Theta(1)$ time complexity.

The experiments were conducted on a Xeon CPU and a Xeon Phi. We evaluated the difference between those platforms, and we also compared the lock-free implementation with different baseline implementations, e.g. Intel TBB concurrent priority queue. The experimental results show that our lock-free implementation is competitive against the others, and it outperforms the baseline implementations in certain operations.

## 1. INTRODUCTION

A priority queue is an abstract data structure, where all the elements in it are associated with a priority key. Therefore, there is an order between all elements which is determined by this key. Priority queues have two main operations: *push* to insert elements and *pop* to dequeue the element with the highest priority. The priority queue guarantees that the element dequeued by a *pop* operation has the highest priority. Elements of equal priority are usually allowed and do not pose a problem. A number of well known applications (e.g. job scheduling, constraint systems, Dijkstra's algorithm, and encoding algorithms) use a priority queue. Hence, many different single-threading implementations have been proposed.

In recent years, CPU architecture have moved from single core to multi core systems. Therefore, the research community has focused on parallelization of single-threading applications and the implementation of efficient concurrent data structures. In a multi-threading application, multiple threads might access a priority queue simultaneously. To guarantee consistency, the data structure has to be thread-safe. One way to achieve thread-safety is mutual exclusion through structures e.g. locks, semaphores and other primitives. All these methods cause blocking and only a single thread, the one in the critical region, might make progress. Consequently, this approach limits scalability.

Another way to achieve thread-safety are lock-free data-structures which guarantee that no thread is blocked and at least one thread makes progress at any time. Apart from a high implementation complexity and challenging correctness verification, this approach is in theory superior to naive mutual exclusion. Atomic synchronization primitives which are necessary for a lock-free implementation are available on most modern x86 CPUs.

In this paper, we present a lock-free concurrent priority queue which exploits the characteristics of the underlying data structure. A skip list [1] is appropriate for a lock-free implementation thanks to its simplicty. It provides probabilistic balancing using multiple levels, maintaining an ordered list of keys which provides fast *pop* operations.

A heap-based concurrent priority queue which is designed for CUDA data parallel SIMT architecture is presented in [2]. The authors use wide heap nodes in order to support thousands of push and pop operations at the same time. Sundell et al. [3] present a lock-free concurrent priority queue which uses a skip list, as the underlying data structure. Their implementation was evaluated on Pentium II architecture using up to 30 threads. In this paper a similar lock-free implementation is evaluated on a state-of-art CPU and a MIC architecture.

This paper is structured as follows. Section 2 gives a description of priority queues and skip lists. Our implementation is presented in detail in section 3. Section 4 describes experimental results collected from a Xeon CPU and Xeon Phi, by comparing the lock-free implementation with the Intel Threading Building Blocks (TBB) concurrent priority queue, and two different implementations using mutual exclusion. Finally, section 5 concludes this work.

## 2. BACKGROUND

The implementation of a priority queue can be achieved using different underlying data structures. A common implementation is a binary-heap which is based on an array. It requires $\mathcal{O}(\log n)$ time complexity for *push* and *pop* operations, and has a space complexity of $\mathcal{O}(n)$ .

Two baseline implementations are used for comparison with our work. The *priority_queue* from the C++ standard library (STD) and the *concurrent_priotiy_queue* implemented in the Intel Threading Building Blocks (TBB) library. The TBB implementation is based on concurrent vectors, while the STD implementation uses vectors (by default) as the underlying data structure. For the experimental evaluation presented in section 4, the STD implementation was made thread-safe, by adding a single mutex which serializes every access to the data structure.

Our lock-free implementation is based on a skip list [1]. A skip list is a multi-level list where the list on the bottom level is equal to a single-linked list. The upper level lists have fewer nodes, in the sense that some nodes are skipped, thereby reducing the time to traverse them. Every node is inserted into the bottom level and also linked the next level with a probability of 0.5. A node is reachable in level 2 with probability 0.5, in level 3 with probability 0.25, and so on. By traversing from the top level list to the bottom one, this structure serves as an index which is asymptotically equal to a binary tree. Therefore, every node on the bottom level can be accessed in $\mathcal{O}(\log n)$. Since the nodes are sorted, the *pop* operation which removes the node with the highest priority, takes only $\Theta(1)$.

On average, every node has two pointers to other nodes. This is the same size as a node in a binary tree or heap. Consequently, the memory consumption of a skip list is similar.

A lock-free skip list is a good choice for a concurrent priority queue, since the atomic operations are only involving a few nodes, and the probability of conflicts between threads is minimized.

## 3. PROPOSED APPROACH

A high-level description of a skip list has already been introduced in the previous section. The implemented skip list is similar to the ones proposed by [4, 3]. While [4] uses Java and its *AtomicMarkableReference*, we used C++11 and its atomic operations. Since C++11 does not provide an atomic markable reference object, we implemented our own, such that we can check a change of the reference or the mark in a single Compare-and-Swap (CAS) operation. For obtaining such a reference, multiple implementation alternatives are possible: using a bit of the pointer as a mark, double CAS, double-width CAS, and Transactional-Memory.

For our concurrent priority queue, we implemented an *AtomicMarkableReference* object, by using the least significant bit as a mark. Thereby we are able to use the C++11 *compare_and_exchange* operation and are not relying on double CAS or Transactional-Memory which are less common hardware features. In general, using the least significant bit as a mark does not lead to any issues since most 64 bit processors only allow read or write operations from addresses which are a multiple of 8 bytes. As consequence the less significant bit is zero and unused.

In addition to the common skip list methods (e.g. empty, size, find, remove, and insert), we added a *pop* method, which is required for a priority queue [4], and also batch processing methods for pushing or popping *k* elements.

The implementation for *pop* is straightforward in our concurrent priority queue, since the element with the highest priority is always located in the beginning of the skip list. Therefore a batch *pop* operation has a complexity of $\Theta(k)$. This means no significant performance improvement can be achieved over the regular *pop* operation. On the other hand a batch push operation can be implemented which has some benefits over the single element operation. Since the batch push operation assumes that the input is already sorted, it is able to insert ranges of the input array instead of single elements. Thereby the number of atomic operations is reduced. In the worst case, when each range holds only one element, the batching operation behaves similar to the regular push operation.

### 3.1. Correctness verification

Two different applications were used for verifying the correctness of our concurrent priority queue. A lossless data compression algorithm and a shortest path algorithm. First, the Huffman encoding algorithm was used to test the correctness for a single-threaded execution. The algorithm was implemented in a way that can use either an existing min-heap implementation or our own priority queue implementation. To verify correctness, the outputs of the two variations were compared for a variety of inputs.
A second correctness test was based on a concurrent shortest path algorithm which is provided as an example with the Intel TBB source code. It originally utilizes the TBB concurrent priority queue, but we adapted the algorithm to also use our own lock-free implementation. In comparison to the previous test, this application accesses the priority queue concurrently. Again the outputs for different input graphs of the two versions were compared to verify correctness.

Performance benchmarks for these algorithms were not conducted, since it might have implied tuning the algorithms themselves. Such optimizations are out of the scope of this project. In section 4, performance evaluation against TBB concurrent priority queue, is presented for specific workloads.

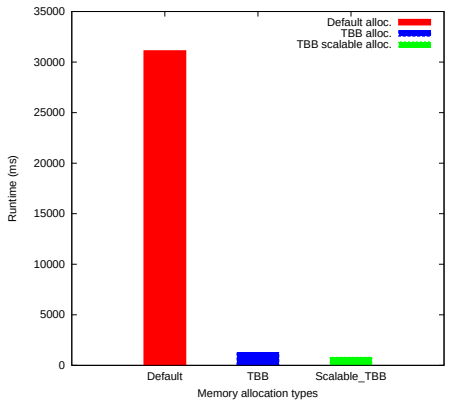



**Fig. 1**: Runtime allocating 10 Mio elements using C++ default, TBB default and TBB scalable allocator.

### 3.2. Memory allocation micro-benchmark

Early overall measurements indicated that memory allocation on a Xeon Phi might represent a significant part of the runtime. Therefore, we performed a memory allocation benchmark on the Xeon Phi. Three different allocators were considered: the default C++ allocator, the default TBB allocator and the TBB scalable allocator.

TBB's scalable allocator is worth describing as many external applications can benefit from it. This allocator uses a global memory heap, and provides each thread its own memory heap in order to reduce the amount of code for synchronization, and for avoiding false sharing. Then each thread accesses its own memory heap through thread-specific data, using system APIs. The allocator gets 1 MB chunks from the operating system and divides them into 16 KB aligned blocks. Then, it initially places these blocks in a global heap of free blocks. Memory request from the application get served from this heap, if the thread private heap does not hold any more free blocks. In addition to that, the global heap re-uses memory blocks, by keeping freed ones from the application instead of returning them directly to the operating system. In case a thread does not find free objects within its own heap and there are no available blocks in the global heap, additional blocks are requested to the operating system [5, 6].

Our micro-benchmark consists of allocating ten million skip list nodes with 240 threads and measuring the time it takes to accomplish this task on the Xeon Phi. As expected, the C++ default allocator has a much higher runtime than the two TBB allocators. These two allocators obtained a 24x and 38x improvement, respectively, over the C++ default allocator. The results are displayed in Fig. 1.

While running this micro-benchmark, we observed that the core running the operating system shows a very high utilization which means that if many cores want to allocate memory they are also bound by the single core hosting the operating system.

## 4. EXPERIMENTAL RESULTS

### 4.1. Evaluation on Workloads

The experiments were conducted on two different systems. First, a CPU-based system equipped with an Intel Xeon E3-1245 processor running 4 cores at 3.4 Ghz and 24 GB main memory and using Ubuntu 14.04 as operating system. Second a MIC, the Intel Xeon Phi 7120P which has 60 cores with 4 hyper-threads, each clocked at 1.23 Ghz. It has 16 GB main memory with a bandwidth of 352 GB/s using CentOS 6.5 as host operating system. The code was compiled using Intel C++ compiler (*icpc*) version 15.0.0 for both systems. To allow a comparison, the experiment parameters used were the same on both platforms. For each experiment five iterations were averaged.

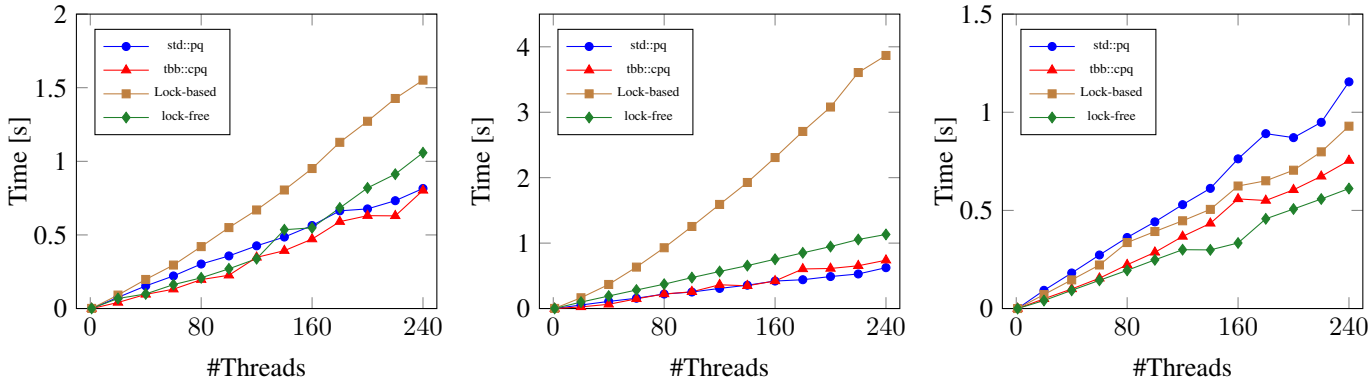
Three types of workload were executed, *mixed*, *push* and *pop*. For the mixed and pop workloads, the concurrent priority queue was prepopulated with 1000×threads for the former and 10 Mio. elements for the latter. Prepopulation was not included in the measured runtime. The *mixed* workload chooses between a *push* or *pop* operation with a probability of 50%. The number of threads is increased from 1 to 240, in steps of 20, while the number of operations per threads stays constant.

#### *4.1.1. Evaluation on Xeon Haswell*

Fig. 2 shows the run-times for executing the three workloads on a Xeon E3-1245 system. All variants are performing and behaving similar in the *mixed* and *push* workload, except for the lock-based implementation which struggles more to scale with the number of threads. Our lock-free implementation is very close to baseline implementations and shows similar behavior when increasing the number of threads. In the *pop* workload, all four data structures behave very similar. The lock-free implementation of the pop operation is done in $\Theta(1)$ and lazy-deletion shows the best runtime by a small margin.

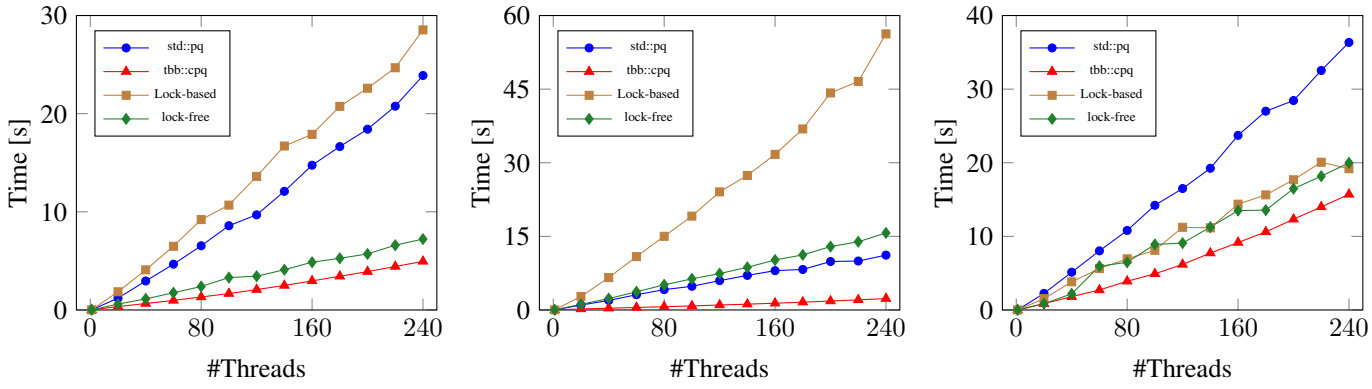
#### *4.1.2. Evaluation on Xeon Phi*

The same workloads were also run on Xeon Phi. The measured run-times are plotted in Fig. 3. In the mixed workload the TBB and the lock-free concurrent priority queues behave similarly well. The other two implementations show clearly a longer runtime. Looking at the *push* wrt. *pop* workload it is visible that the longer runtime in the lock-based implementation is originated by the poor performance




(a) Mixed workload (b) Push workload (c) Pop workload

**Fig. 2**: Runtime for different workloads executed on a Xeon E3-1245 while varying the number of threads




(a) Mixed workload (b) Push workload (c) Pop workload

**Fig. 3**: Runtime for different workloads executed on a Xeon Phi while varying the number of threads

from the *push* operation. For the STD implementation we used a lock-based wrapper which leads to a long runtime for the *pop* operation. The TBB implementation behaves well in all three workloads. Our lock-free implementation shows almost the same behavior with a slightly longer runtime.

Comparing the runtime of the two platforms shows that the same workload is about one magnitude faster on Xeon CPU.

### 4.2. Impact of cache-invalidation

The previous experiments show that the same workload performs roughly a magnitude better on a Xeon CPU than on Xeon Phi. Despite the fact that Xeon Phi has many more cores and a higher memory bandwidth. There are multiple reasons for these, e.g. cache structure, memory allocation cost, and other problems. Therefore an experiment was designed to check how Xeon Phi performs, in comparison to other architectures when cache-invalidations are not occurring. This consists of prepopulating our lock-free concurrent priority queue with 10 million elements, then every thread performs 100'000 element lookups. Thereby the structure of the priority queue is unaltered which means no cache-line belonging to the concurrent priority queue should get invalidated. This experiment was executed with 1 up to 240 threads. The run-times from executing this experiment on an Intel Core i7-3820, a Xeon CPU and the Xeon Phi, are plotted in Fig. 4. It can be noted that the runtime on Xeon Phi stays stable up to 60 threads equaling the number of cores. Even with more threads its runtime increases only slightly. Whereas on the other two systems, the execution time increases linearly. The results show clearly that the invalidation of cache-lines and their consequences have a major impact on the performance on Xeon Phi. While in the previous experiments it showed a linear increase in execution time when scaling up, in this case the runtime stays

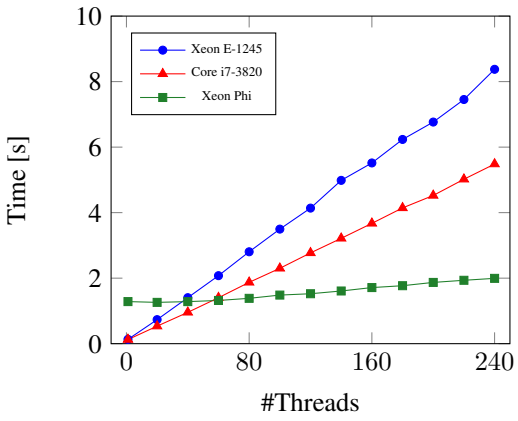



**Fig. 4**: Cache-invalidation assessment

almost constant which is close to ideal.

This finding is not new, other researchers [7] have shown that a drawback when using a Xeon Phi is that any cache-line invalidation might lead to many cores ending up with invalid cache lines . This is due to fact that it uses a distributed tag directory. Thus, the more hardware cores are used on a concurrent application the probability that at least two threads share a cache line is higher. These dirty cache lines produce expected overhead due to cache invalidations.

Implementing a truly concurrent data structure is a challenging task for many reasons. For instance, even though our implementation is lock-free, it may not be starvation-free. There could be a thread A that when going through the lowest level of the skip list searching for the next un-marked node (i.e. logically undeleted) gets always outrun by some other thread B. This means that thread A can fail repeatedly if the others threads always succeed. Moreover, thread contention may happen if many threads try to logically delete a node (i.e. mark a node). Only one thread will succeed and all the other unsuccessful ones will race to mark the next available node.

### 4.3. Operational intensity in different microarchitectures

Operational intensity is defined as the ratio of the number of instructions executed to the number of memory accesses [8]. If there exist many instructions per memory access, then the program is considered to have a high computational intensity i.e. compute bounded. On the other hand, if there are a small number of instructions are executed per memory access, then the program is considered to have a low computational intensity i.e. memory bounded.

Our project goal was to design a simple, yet effective, concurrent priority queue. Thus, we expected to have an operational intensity dominated mainly by the number of memory accesses, and aimed to improve this. Having to move data around, has a different impact on different CPU architectures. We will describe and explain how our data structure behaves on Intel Haswell microarchitecture (Intel Core i7-4558U) and on Sandy BridgeE microarchitecture (Intel Core i7-3820K).

The Intel Haswell microarchiteture is the successor of Ivy Bridge which in turn is the successor of Sandy Bridge. They have several differences but they also share many commonalities. One of the biggest change is the enhancement done on cache level operations. These changes are summarized in Table 1 [9, 10, 11].

| Metric | Sandy BridgeE | Haswell |
|---|---|---|
| L1 Load Bandwidth | 32 Bytes/cycle | 64 Bytes/cycle |
| L1 Store bandwidth | 16 Bytes/cycle | 32 Bytes/cycle |
| L2 Bandwidth to L1 | 32 Bytes/cycle | 64 Bytes/cycle |
| L2 Unified TLB | 4K:512, 4-way | 4k+2M shared: 1024, 8-way |

**Table 1**: Cache operation differences between Intel Haswell and Intel SandyBridgeE

In addition to core cache size, latency, and bandwidth improvements, the Intel Haswell microarchitecture has also improved its ICache prefetch algorithms, and the way it handles conflicts. It uses hardware transactions i.e. it uses hardware to keep track of which cache lines have been read from and which have been written to. L1 cache tracks addresses read/written from/to respectively in the transactional region and it may evict address but without loss of tracking. Data conflicts occur if at least one request is doing a write, but it is detected at cache line granularity using existing cache coherence protocol [12, 13].

While running our priority queue benchmark on these two different architectures, we observed different behaviors. The running times when using an Intel IvyBridge processor dramatically increase due to a higher number of instructions and cache misses. Everytime we need to perform an insertion, we first have to search for the adequate place within the skip list. The average of the skip list nodes fit in a 64-byte cache line but the ones containing pointers in upper levels do not. On the other hand, when we used an Intel Haswell processor, running times were much less than in the Sandy BridgeE processor. This is mainly due to the improvements done on cache operations. In this case, our data structure can take advantage of such improvements by loading more skip list nodes into the caches that can also be used by other threads.

Fig. 5 shows how operational intensity behaves when running different amounts of insert operations over such micro-architectures. It can be noted that when performing a small number of operations, our data structure is CPU bounded on an Sandy BridgeE processor, but memory bounded on a Haswell processor. This is because in the former we have a small number of cache-misses against a really high number of instructions whether in the latter we observed

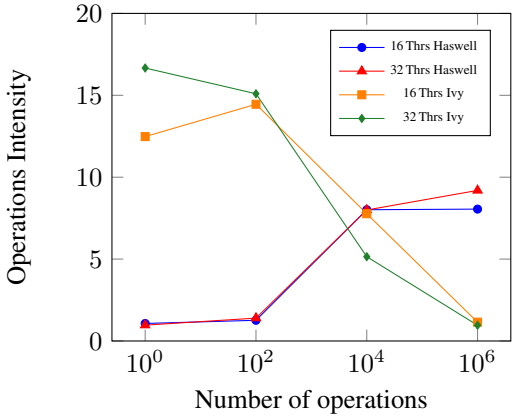



**Fig. 5**: Op. Intensity in Intel Haswell and Intel Sandy BridgeE microarchitectures

a low operational intensity because we need less number of instructions for performing such tasks. When we increase the number of operations, the Sandy BridgeE processor gets many more cache-misses compared to the Haswell one. Thus, in the former one our data structure becomes memory bounded and in the latter one CPU bounded.

## 5. CONCLUSIONS

We presented a lock-free concurrent priority queue based on a skip list. The implementation was evaluated on a common x86 CPU and more interesting on a MIC architecture, the Xeon Phi. The results show that our implementation is competitive in comparison to the assessed baseline implementations. On the MIC architecture, all tested implementations are below our expectations because they are unable to take advantage of this modern hardware architecture. Further optimizations specifically for the MIC architecture are necessary to achieve expected performance.

## 6. REFERENCES


[1] William Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, June 1990.

[2] Xi He, Dinesh Agarwal, and Sushil K. Prasad, "Design and implementation of a parallel priority queue on many-core architectures," in *19th International Conference on High Performance Computing, HiPC 2012, Pune, India, December 18-22, 2012*, 2012, pp. 1–10.

[3] Håkan Sundell and Philippas Tsigas, "Fast and lock-free concurrent priority queues for multi-thread systems," *J. Parallel Distrib. Comput.*, vol. 65, no. 5, pp. 609–627, May 2005.

[4] Maurice Herlihy and Nir Shavit, *The Art of Multiprocessor Programming*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[5] "The foundations for scalable multi-core software in intel threading building blocks methodology, tools, and techniques to parallelize large-scale applications: A case study future-proof data parallel algorithms and software on intel multi-core archite," .

[6] Richard L. Hudson, Bratin Saha, Ali-Reza Adl-Tabatabai, and Benjamin C. Hertzberg, "Mcrt-malloc: A scalable transactional memory allocator," in *Proceedings of the 5th International Symposium on Memory Management*, New York, NY, USA, 2006, ISMM '06, pp. 74–83, ACM.

[7] S. Ramos and T. Hoefler, "Modeling Communication in Cache-Coherent SMP Systems - A Case-Study with Xeon Phi," in *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing*. 06 2013, pp. 97–108, ACM.

[8] Roger W. Hockney, *The Science of Computer Benchmarking*, Software, environments, tools. Society for Industrial and Applied Mathematics, 1996.

[9] Jain Tarush and Agrawal Tanmay, "The Haswell Microarchitecture - 4th Generation processor," `http://ijcsit.com/docs/Volume%204/vol4Issue3/ijcsit2013040321.pdf`, 2013, Accessed: 2014-12-15.

[10] Fog Agner, "The microarchitecture of Intel, AMD and VIA CPUs An optimization guide for assembly programmers and compiler makers," `http://www.agner.org/optimize/microarchitecture.pdf`, 2014.

[11] Sadika Amreen Kapil Agrawal Thananon Patinyasakdikul, Reazul Hoque, "Intel Core Architecture. An Analysis of the Haswell and Ivy Bridge Architectures by Intel," `http://web.eecs.utk.edu/courses/fall2013/cosc530/CS530Project_intel.pdf`, 2013.

[12] Rajwar Ravi, "Going under the hood with Intel's next generation microarchitecture codename Haswell," `http://pages.cs.wisc.edu/~rajwar/papers/rajwar_qconsf2012.pdf`, 2012, Accessed: 2014-12-15.

[13] Kanter David, "Intel Haswell CPU Microarchitecture," `http://www.realworldtech.com/haswell-cpu/5/`, 2012, Accessed: 2014-12-15.