FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Tomáš Faltín

Streaming system scheduling for Xeon Phi

Department of Software Engineering

Supervisor of the master thesis: RNDr. Yaghob Jakub, Ph.D.

Study programme: Computer Science

Study branch: Software Systems

Prague 2016

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In …..... date..... signature of the author

Title: Streaming system scheduling for Xeon Phi

Author: Tomáš Faltín

Department: Department of Software Engineering

Supervisor: RNDr. Yaghob Jakub, Ph.D., Department of Software Engineering

Abstract: Task scheduling in operating system area is a well-known problem on traditional system architectures (NUMA, SMP). Unfortunately, it does not perform well on emerging many-core architectures, especially on Intel Xeon Phi. We collected all publicly available information about the architecture of Xeon Phi. After that, we benchmarked the Xeon Phi in order to find the missing information about its architecture. We focused especially on the information about cores and memory controllers. These are the most important parts when designing a scheduler. Based on the results, we proposed improvements for scheduling algorithm in the Bobox (an experimental streaming system). However, we found that the biggest problem is not in the scheduling algorithm, but in the design of operators' parallelization. Therefore, we proposed improvements to the parallelization and tested one of the proposals.

Keywords: scheduling, streaming systems, Bobox, Xeon Phi

First of all, I would like to thank my supervisor RNDr. Jakub Yaghob, Ph.D. for his patience and guidance. Other thanks belong to my family and friends for their love, trust and for the fact that they have always believed in me.

Contents

Introduction

There has always been an effort to execute any program as fast as possible. It applies especially to the applications processing some data. One of the main factors influencing the speed of the applications is the speed of the microprocessor, which is processing the data. People try multiple ways to speed up microprocessors. These days, the microprocessors with multiple cores inside are the most popular solution. Intel took this idea and introduced its many-core architecture and launched Xeon Phi cards.

There is an experimental streaming system called Bobox [1]. It was developed at the Department of Software Engineering of the Charles University. It is an example of a complex application processing some data. While trying to run the Bobox on the Xeon Phi, it did not perform well. There is a performance problem on the Xeon Phi card, even though it is not connected with the traditional system architectures, i.e. SMP (Symmetric multiprocessing) and NUMA (Non-uniform memory access).

In this thesis, we want to find the cause of the performance problems of the Bobox on the Xeon Phi. Firstly, we make a summary of all available information about the architecture of the Xeon Phi. Secondly, we benchmark the coprocessor and analyze the results in order to find some additional information about its architecture. The information are used to a proposal of possible improvements to the scheduler and the memory allocator of the Bobox. Finally, we test the proposed improvements in the Bobox and analyze results. If the improvements do not solve the problems, we will try to find a real cause of the problems. If the problems are found, we will introduce another enhancement to the system.

The first chapter analyzes our performance problem and prepares a working procedure of our research.

The second chapter makes an overview about hardware and software architecture of the Xeon Phi. It is a summary of all public documents we were able to find. There is a thorough description of the hardware architecture. We focus the description on things connected to the core and the memory layout. These components are important when dealing with the problems connected to scheduling. On the other hand, the software architecture contains only some basic information necessary for using the Xeon Phi.

The third chapter is about testing. It describes the preparation of the Xeon Phi and its environment for testing. It writes about the Xeon Phi specific of the testing environment. Afterwards, it continues with the description of the used methodology

and our testing. Based on the tests, we find some missing information about the Xeon Phi hardware architecture. We also use that chapter to propose improvements of scheduling on the Xeon Phi.

The fourth chapter describes the streaming system Bobox and its internals. We try to test the improvements using this application. However, we show that the biggest problems are caused neither by the architecture nor by the scheduler. To prove our speculation, we find the real cause of the problems and test one of the simple improvements.

The final chapter summarizes a contribution of this thesis and proposes a future work.

1. Analysis

To analyze the main problem with scalability of the Bobox, we start with the attempt to replicate the results of the measurement given by [2]. That document focuses on creating a new task scheduler for the Bobox. It takes different effects of a processor into account, i.e. many layers of caches and NUMA effect. We successfully replicated the results on a similar configuration. However, there is a big performance slowdown while running the same tests on Xeon Phi, even though we use different configurations with a different number of NUMA nodes. The best results are achieved while the configuration is set to NUMA.

There are multiple papers [3], [4] or [5] about benchmarking the Xeon Phi. They all try to measure different characteristics of the Xeon Phi, e.g. latency of instructions, throughput of instructions, memory access latency and memory bandwidth. The papers did not mention a NUMA effect at all. Closer look reveals that the experiments would not detect the NUMA effect. We did not find any other papers or researches that confirm or decline the NUMA effect on the Xeon Phi. As the results show a possibility of the Xeon Phi to be a NUMA system, we create our own benchmarks to test the NUMA effect.

We analyze the results and set the configuration of Bobox according to them. However, it does not solve the scalability problem of Bobox. To be able to find a problem, we use a profiling tool – Intel VTune Amplifier [6]. This profiler has a special support for the Xeon Phi cards.

Lots of valuable information about Bobox is stored in [2] or [7]. There are sections covering intra-operator parallelization, task scheduling and memory allocation. As a result of the profiling, we locate the real problems and propose a solution how to deal with them. We also implement one of the solutions as a proof.

2. Xeon Phi overview

There are two types of Intel Xeon processors. One of them is the standard type of server processor, not so different from the classic desktop processors. The other one, with attribute Phi in its name, is compounded of many small processors and its architecture is closer to the architecture used in graphic cards.

The latest architecture, by the time of writing this thesis, is Knights Landing (launched in June 2016). The previous architecture is called Knights Corner. Unless stated otherwise, all things in the thesis relates to the older architecture – Knights Corner.

There are 8 types of the Intel Xeon Phi coprocessor with architecture Knights Corner. The only differences are: a number of cores, processor base frequency, a number of memory channels and the maximal size of used memory. For simplification, we write only about the model Intel Xeon Phi 7120. We choose this model, because we have it at our disposal.

This chapter makes an overview about hardware and software architecture of the Xeon Phi. It is a summary of all public documents we were able to find. There is a thorough description of the hardware architecture. We focus the description on things connected to core, memory layout and interconnect. There are also basic information about software architecture. We cover only necessary things for running applications.

2.1. Hardware Architecture

The coprocessor comprises of 61 in-order cores that are connected through on-die bidirectional interconnect. In addition to the cores, there are also 8 memory controllers and other special devices placed on it, such as memory controller (GBOX), PCI Express client logic (SBOX) or a display engine (DBOX)[1].

As shown in Figure 1, each core contains:

- 512-bit vector processor unit (VPU)
- The Core Ring Interface (CRI)
- The L2 cache
- The tag directory (TD)
- Asynchronous Processor Interrupt Controller (APIC).

[1] Do not confuse with a display device output. The display engine stores debugging information.

Core fetches and decodes the instructions from four hardware thread execution contexts. Vector processor unit executes floating-point and integer operations. The Core Ring Interface hosts L2 cache, APIC, TD and connects each core to the Ring Stop. TD holds coherence between L2 caches of the core. Memory controller is comprised of two independent memory channels where each channel has 32-bit in width and provides connection to the RAM. SBOX includes PCI Express client logic, DMA and the power management capabilities. DBOX supports the debugging of the Xeon Phi.

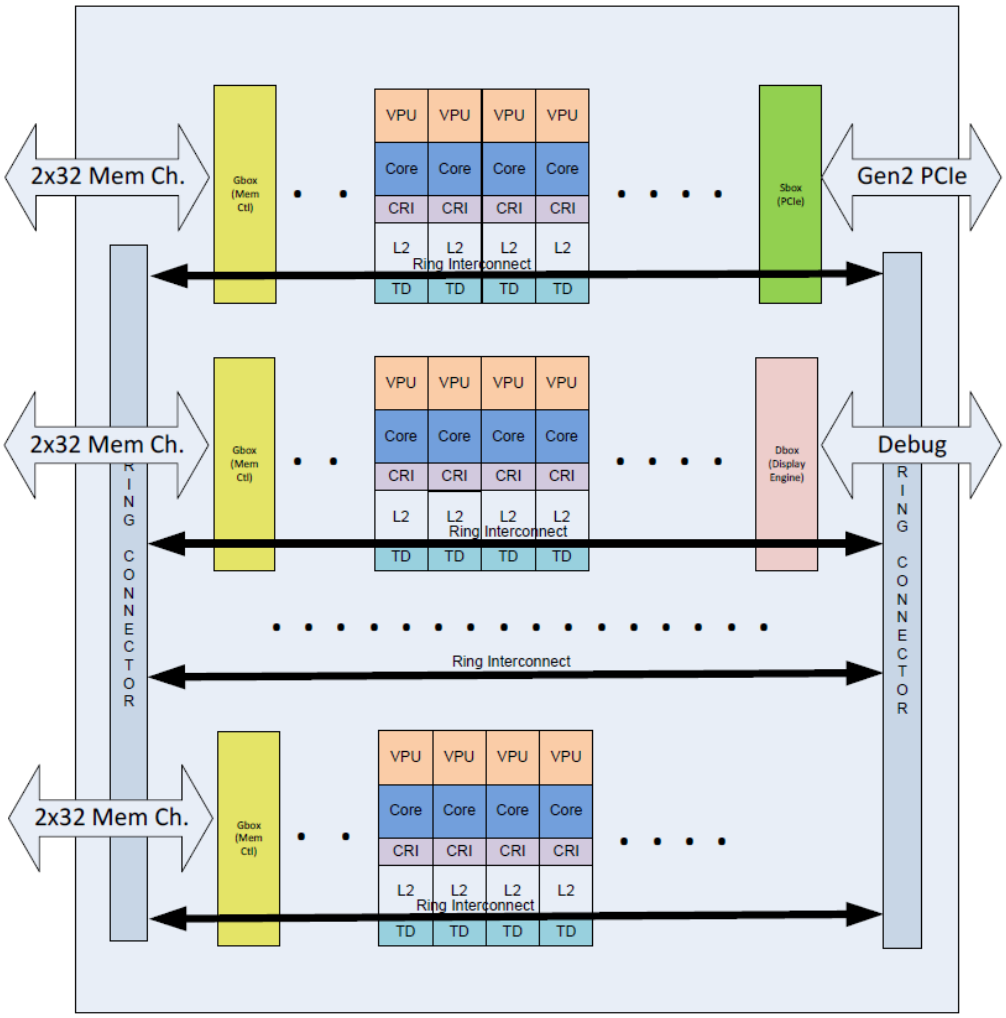


Figure 1: Basic building block of the Intel Xeon Phi (source [8])

2.1.1. Core

Xeon Phi core design is based on the old Pentium P54c architecture (see documentation [9] for architecture details). There were made some additional changes to support hardware threads.

Each core contains four hardware threads. Threads support 32-bit and 64-bit execution environment. Core also contains 8-way 32KB L1 ICache (instruction

cache), DCache (data cache) and some interfaces with L2/CRI block. Each thread has a replicated architecture state which comprises of the registers (GPRs, ST0-ST7, segment registers, CR, DR, EFLAGS), the prefetch buffers, the instruction pointers, the segment descriptors and the exception logic. Other additional adjustments, added to support hardware threads, are thread ID bits in TLBs, thread-specific flush and a hardware support for thread wake-up/sleep mechanisms.

Pipelines

The core pipeline is composed of 7 stages. Pipeline for vector instructions adds other 6 addition stages. Both pipelines use global stall architecture. It means that the part of the pipeline stalls if a stage is stalled. Each core stage is speculative – it might invalidate the whole work in a pipeline if a branch misprediction appears.

Figure 2 shows the core pipeline. Figure 3 shows the vector pipeline, where orange boxes show addition stages in vector pipeline compared to the core pipeline. Figure 4 depicts the core pipeline of Xeon Phi compared to the generic one. Table 1 shows the function of each stage.

Figure 2: Core pipeline

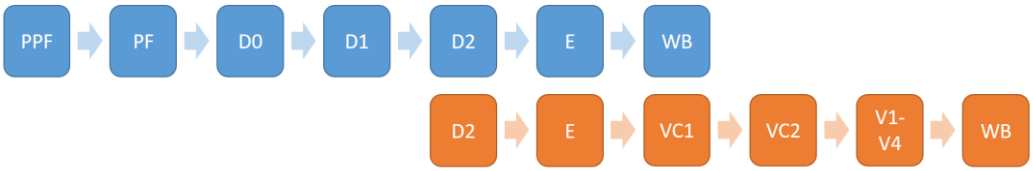


Figure 3: Vector pipeline

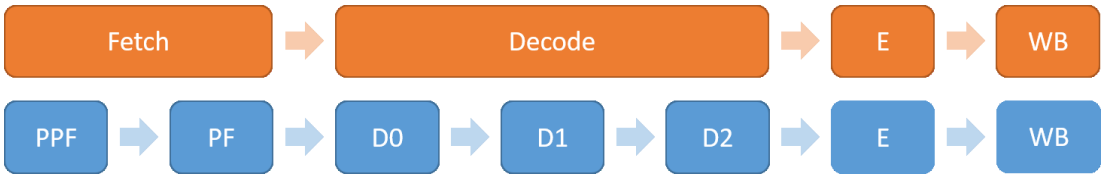


Figure 4: Core pipeline compared to the generic one

Stage	Work
PPF	Thread picker
PF	Instruction cache lookup Prefetch buffer write
D0	Thread picker Instruction rotate Decode of 0f, 62, D6, REX prefixes
D1	Instruction decode CROM lookup Sunit register file read
D2	Microcode control execution Address generation Data cache lookup Register file read
E	Integer ALU execution Retire/stall/exception determination
WB	Integer register file write Condition code (flag) evaluation

Table 1: Core pipeline stages functions

Instruction Fetch

Instruction fetch is split into two stages – the PPF (pre-thread picker function) and the PF (picker function). The goal of the stages is to choose which thread is going to be executed. The PPF stage prefetches instructions for a thread context into the prefetch buffers. There are 4 prefetch buffers per a thread and each of them can store 32 bytes in a buffer. Also, there are two instruction streams per a thread. If one of the streams is stalled due to a branch misprediction, the second stream is switched in, while the branched target stream is prefetched. The PF examines the prefetch buffer to determine the next thread. A priority to refill the prefetch buffer is given to a thread which is being executed in the current cycle.

Each core contains a ready-to-run buffer that consists of two instruction pairs. If the executing thread has a control transfer to instruction that is not in the buffer, the context buffer is flushed and appropriate instruction is loaded. In case that the instruction cache does not have a control transfer point, the core stall happens. To hide a performance penalty during the core stall, the PF chooses a next context to execute. It works in a round-robin manner. If the instructions from context 0 are processed in N-th cycle, the PF tries to issue the instructions from context 1, context 2 or from context 3 in (N+1)-th cycle. This exact order is always used.

The two-cycle instruction fetcher unit implies that fully pipelined core cannot issue instructions from the same context in back-to-back cycles. This means that if the instructions from context 0 are issued in N-th cycle, the instructions from all contexts

except for context 0 can be issued in the next (N+1)-th cycle. It implies that there need to be present at least 2 threads for full core utilization. If they are not presented, the core utilization in a maximum of 50% can be achieved due to inability to issue instructions in back-to-back cycles.

The refill of instruction buffer, which happens during core stall, takes 4-5 clock cycles. This implies that 3-4 threads are needed for optimal performance. On the other hand, if the PPF and the PF are perfectly synchronized, two threads are sufficient for running the maximal speed. In case they are not synchronized (due to a cache miss), a cycle clock bubble might be inserted.

An overview of the first two stages is shown in Figure 5. Another view on the instruction and data flow is depicted in Figure 6.

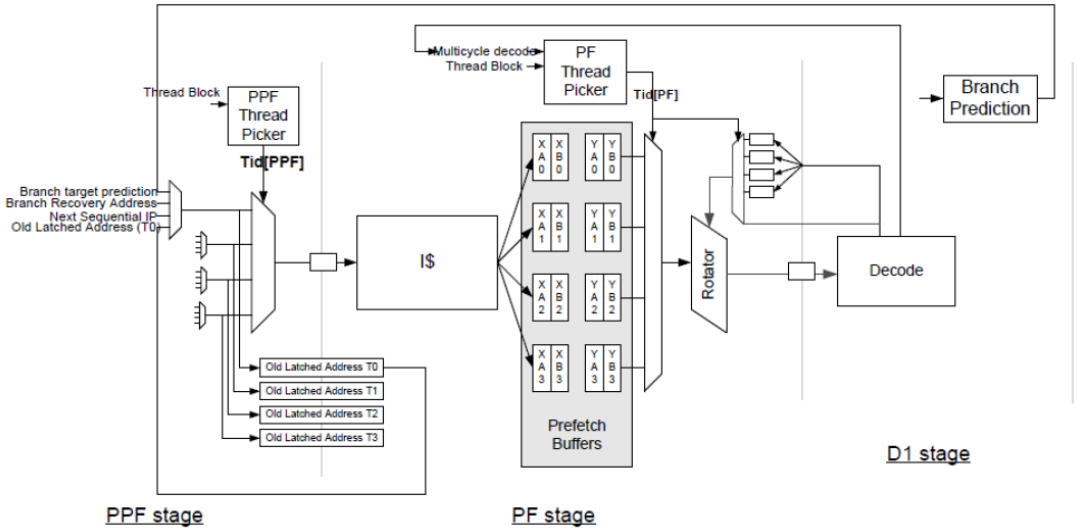


Figure 5: Core multithreading support (source [8])

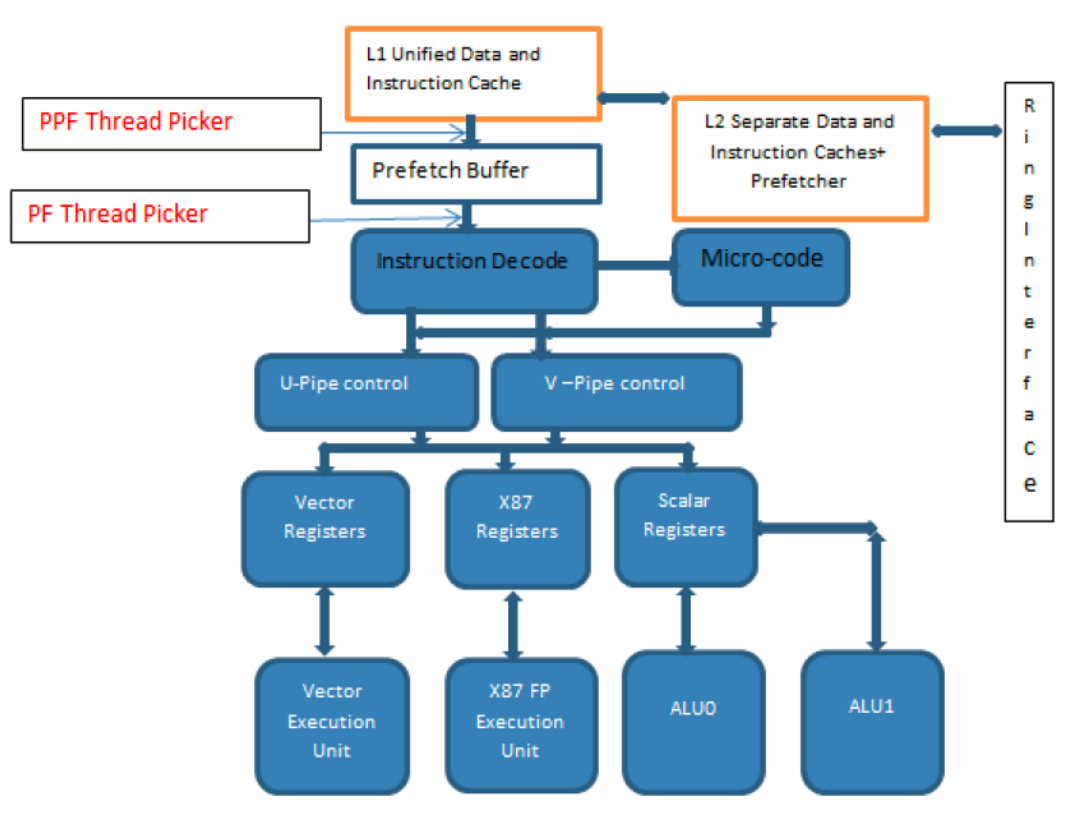


Figure 6: Simplified data and instruction flow (source [10])

Instruction Decode

The decode stages (D0 and D1) start to decode instructions right after the appropriate instructions are chosen by the PF. The instructions are decoded in two cycle clock. The D0 stage decodes the instruction prefixes. Fast prefixes are decoded without penalty, legacy prefixes have 2 clock cycle latency. The D1 stage mixes together the ucode of the decoded prefixes with the instruction microcode found in ucode ROM. In the next decoding stage (D2), the processor reads the general purpose register file and looks up the data cache.

In the end, the decoded instructions are sent to an execution unit by two pipelines – U-pipe and V-pipe. The pipelines use the same names as in old Pentium. The first instruction takes U-pipe and the other one takes V-pipe, if the instructions are pairable. The concurrent execution of instructions is governed by pairing rules. Intel documentation [8] says: "*The V-pipe cannot execute all instruction type, and simultaneous execution is governed by pairing rules. Vector instructions can only be executed on the U-pipe.*" Another Intel document [4] says: "*Most of the VPU instructions are issued from the core through the U-pipe. Some of the instructions can be issued from the V-pipe and can be paired to be executed at the same time with instructions in the U-pipe VPU instructions.*" The second source implies that not all

vector instructions are forbidden on V-pipe. The document also gives the list of instructions that run on V-pipe. These instructions are shown in Table 2.

Instruction Execute and Retire

At the execute stage, integer instructions are executed in the ALUs (Arithmetic Logic Unit). Once the scalar instructions reach the WB (write-back) stage, they are finished.

Type	Instruction
Vector Mask	JKNZ, JKZ, KAND, KANDN, KANDNR, JKNZ, JKZ, KAND, KANDN, KANDNR, KMERGE2L1H, KMERGE2L1L, KMOV, KNOT, KOR, KORTEST, KXNOR, KXOR
Vector Store	VMOVAPD, VMOVAPS, VMOVDQA32, VMOVDQA64, VMOVGPS, VMOVPGPS
Vector Packstore	VPACKSTOREHD, VPACKSTOREHPD, VPACKSTOREHPS, VPACKSTOREHQ, VPACKSTORELD, VPACKSTORELPD, VPACKSTORELPS, VPACKSTORELQ, VPACKSTOREHGPS, VPACKSTORELGPS
Vector Prefetch	VPREFETCH0, VPREFETCH1, VPREFETCH2, VPREFETCHE0, VPREFETCHE1, VPREFETCHE2, VPREFETCHENTA, VPREFETCHNTA
Scalar	CLEVICT0, CLEVICT1, BITINTERLEAVE11, BITINTERLEAVE21, TZCNT, TZCNTI, LZCNT, LZCNTI, POPCNT, QUADMASK

Table 2: Pairable instructions

Vector Pipeline

There is a separate pipeline for x87 floating point and vector instructions that starts right after the core pipeline. Even though the vector instruction reaches WB stage, it is not finished. The core pipeline is supposed to think so, but the vector unit keeps working until it goes through the whole vector pipeline. No exceptions can be raised. Instead, the *VXCSR* flag is set to indicate an exception e.g. underflow, overflow.

Vector Processing Unit (VPU)

There is no support for extended or vector operations i.e. MMX, AVX or SSE known from some other Intel processors. This led up to the introduction of new vector floating-point unit (VPU) together with the new instructions.

VPU is 512-bit wide and is placed in each core. It executes majority of instructions in 4 clock cycles with 1 cycle throughput. It can read/write one vector from/to memory. This vector can contain either sixteen 32-bit numbers (integer, single-precision floating point) or eight 64-bit numbers (64-bit integer, double-precision floating point).

VPU is composed of 8 UALUs where each of them contains 2 SP (single precision) and 1 DP (double precision) ALUs with the independent pipelines. Each VPU has 128 entry vector registers divided equally among all 4 threads, which makes

it 32 registers per a thread. There are additional 8 16-bit mask registers controlling which elements are active during computation. Each VPU instruction must pass through one of the five pipelines to completion. The pipelines and their relation to the vector pipeline stages is shown in the Figure 7. It shows which stages execute which pipelines. Possible vector pipelines are:

- **Double Precision (DP) Pipeline** – executes float64 arithmetic, conversion and comparison
- **Single Precision (SP) Pipeline** – executes float32/int32 arithmetic, logical operations, loads (including int64 load), conversion, float64/int64 logical
- **Mask Pipeline** – executes mask instructions with 1 clock cycle latencies
- **Store Pipeline** – executes store instructions with 1 clock cycle latencies
- **Scatter/Gather Pipeline** – read/write sparse data from/to vector registers

A transition between pipelines costs some additional clock cycles. There are built-in bypasses in pipelines, but there are no bypasses between SP and DP pipeline. That is why the execution of the SP instructions consecutively results in good performance unlike the mixing SP and DP instructions, as a result of the fact that there are no bypasses between those two ALUs.

The D2 stage decides whether the processor is processing a vector instruction or a scalar instruction. The E stage detects any dependency stalls. The next two stages (VC1 and VC2) do the shuffling and load conversions. Main add and multiply operations are executed it the V1-V4 stages. The last stage (WB) writes the content of vector/mask register back to cache. Detailed description of vector processing unit is depicted in the Figure 7.

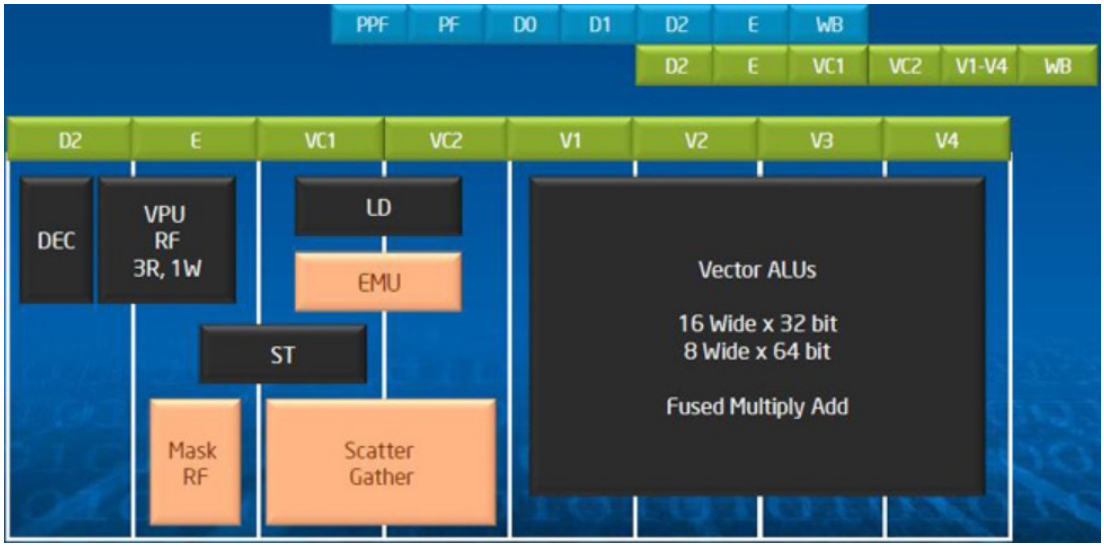


Figure 7: Vector processing unit (source [11])

Latencies

Each stage of the pipeline is processed in 1 clock cycle. If there are at least 4 independent instructions, the pipeline can throughput one instruction per clock cycle with each instruction latency of 4 clock cycles. If there are data dependencies, for example in two consecutive SP instructions, the second instruction must wait until the data is produced by the V4 stage and pass the data through internal bypass to the V1 stage of the next instruction. This causes an additional 3-cycle delay, because there are 3 stages between V1 and V4, where the dependent instruction cannot execute, until it has the data from the previous instruction.

If an SP instruction is followed by another SP instruction with register swizzle, the data has to be send from V4 to VC1 stage by another internal bypass. This causes the additional 5-cycle delay. If an SP instruction is followed by an EMU instruction, since transcendental lookup happens in VC1, the data can be moved by an internal bypass with the 5-cycle latency delay.

If a DP instruction is followed by a SP instruction, the DP instruction has to complete write before the dependent SP instruction can execute. Because there are no bypasses between the SP and the DP pipelines, the waiting costs 7 clock cycles.

Vector Registers

Each thread has 32 512-bit general vector registers *zmm0-zmm31*, eight 16-bit mask registers *K0-K7* and the status register *VXCSR*. The vector registers operate either with sixteen 32-bit elements or with eight 64-bit elements. The *VXCRS* holds the status of each vector operation. The VPU reads and writes the data cache at a cache-

line granularity of 512 bits (64 bytes) through dedicated 512-bit bus. The data read from the cache goes through the load conversion and swizzling to ALU. The writes go through conversion and alignment right into a write-commit buffer in the data cache.

The vector mask registers control updates of the vector registers in the calculation. A small set of operations can be performed on the mask registers – *xor, or* and *and.* Non-masked operations can simply overwrite the destination register by the result of another operation. The masked operations update only the destination register according to the vector mask register. A write mask modifier can be specified with all vector operation. The destination elements that are flagged with '*1*' are updated, the elements with flag '*0*' are left unchanged. If no mask is specified, the default mask *0xFFFF* is used. This value is stored in the mask register *K0*.

Extended Math Unit (EMU)

The VPU contains the Extended Math Unit (EMU). The EMU executes the single-precision transcendental operations using Quadratic Minimax Polynomial approximation and table lookup for fast approximations to the approximation function. The EMU is fully pipelined. The table lookup happens in parallel. The EMU offers following basic transcendental functions: reciprocals, reciprocal square roots, base 2 exponential and base 2 logarithms. Other functions – division, square root and power – are composed of the basic transcendental functions. Table 3 shows latencies of the EMU instructions.

Instruction	Latency [cycles]	Throughput [cycles]
Exp2	8	2
Log2	4	1
Recip	4	1
Rsqrt	4	1
Power	16	4
Sqrt	8	2
Div	8	2

Table 3: EMU instructions latencies

2.1.2. Memory

Figure 8 shows the distribution of the memory controllers. As we can see, each core contains L2 cache. The *memory controllers (MC)* and tag directories (TD) are placed symmetrically around the bidirectional ring.

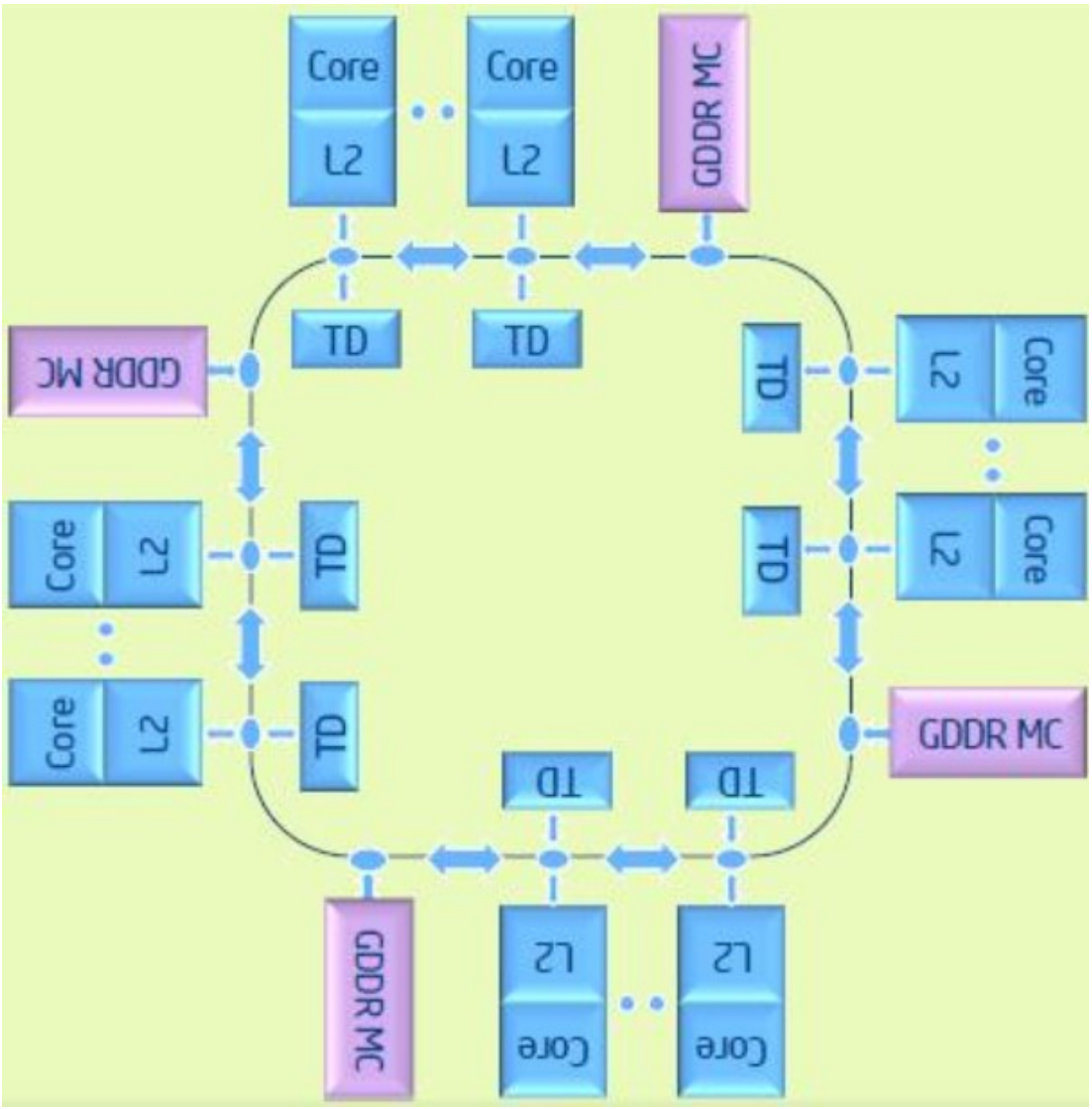


Figure 8: Interleaved memory access (source [11])

L1 Cache

Each core contains a L1 cache. It is composed of a 32KB instruction cache (ICache) and a 32KB data cache (DCache). It has standard 64 bytes cache line size with 8-way associativity. The data from the cache can be returned out-of-order. Load-to-use latency is 1 cycle for scalar integer values, but it is longer for vector values. The cache contains an address generation interlock with minimal 3 clock cycle long latency. Therefore, the GPR registers must be produced by three or more clocks before they are used as a base register or as an index register in an address computation. It implies that the base and index register setup takes also a 3 clock cycle latency.

L2 Cache

The L2 cache is inclusive of the data and the instructions from the L1 cache. It implies that all data that are stored in the L1 cache are duplicated in the L2 cache. The

size of L2 cache is 512KB per core. It is divided into 1024 sets with 8-way associativity per a set and 64 bytes per a way. It is also split into 2 banks and it is able to address 32GB (35 bits). The latency of the cache is 11 clock cycles and the idle access time is around 80 clock cycles. It cannot be accessed by each clock cycle (like L1 cache), but only every other clock cycle.

All cores together contain around 30MB of the L2 cache – each core provides its 512KB. This is valid only if the data among cores are not shared. In this case, each core has only its private data. If the data are shared, they need to be replicated to each core which needs them. In the worst case scenario, when all data are shared among all cores, the effective size of L2 cache is only 512KB.

The cache contains streaming hardware prefetcher that is able to prefetch code, read and RFO (read for ownership) cache lines into the cache. There are 16 memory streams that can deliver together up to 4KB page of data per a request. If the direction of the stream is detected, 4 multiple prefetch requests can be issued.

The L2 cache supports ECC and multiple power states e.g. C1, C6 and the package C3. It is a part of the CRI (core ring interface). It forms the Transaction Protocol Engine together with the TD (tag directory) and the RS (ring stop). This is an equivalent to the FSB unit.

Caches

Both caches (L1 and L2) use a pseudo-LRU algorithm for replacement. They both are also fully coherent. They support 38 outstanding (read/write) requests per a core. The system agent (containing PCI Express agent and DMA controller) can generate additional 128 requests (read and write). Each request executes in 1 clock cycle. The maximal number of requests is $38 \times (\text{number of cores}) + 128$. This allows the software to prefetch data aggressively. If the maximal number of requests is reached, the new request causes a core stall.

The standard MESI protocol [8] is used for maintaining shared states across the cores. There is the state diagram of the MESI protocol in Figure 9. The definitions of the states of the protocol are in Table 4.

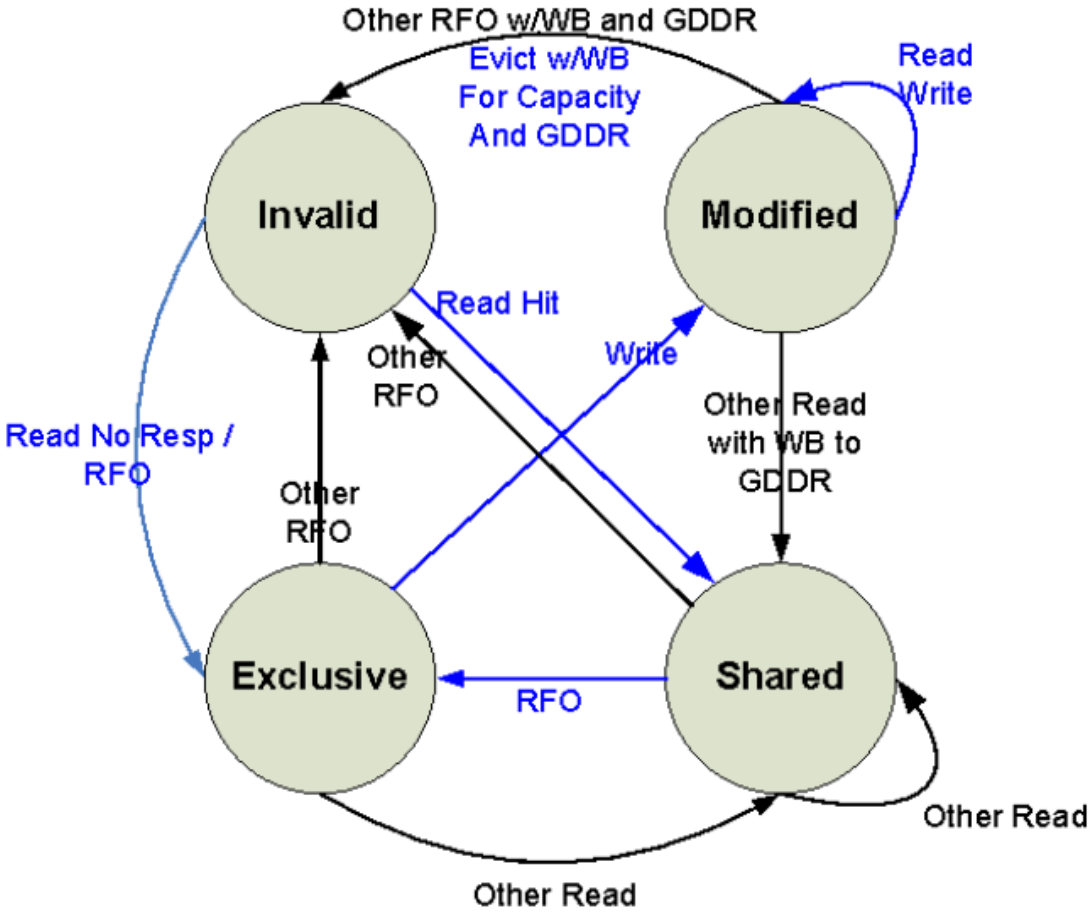


Figure 9: MESI protocol (source [8])

Cache state	State definition
M (Modified)	Cacheline is updated relative to memory (GDDR). Only one core may have a given line in M-state at a time.
E (Exclusive)	Cacheline is consistent with memory. Only one core may have a given line in E-state at a time.
S (Shared)	Cacheline is shared and consistent with other cores and with memory. Multiple cores may have a given line in S-state at the same time.
I (Invalid)	Cacheline is not present in this core cache.

Table 4: Cache states

Distributed Tag Directories (DTD)

To solve potential performance limitations of the MESI protocol in comparison to the MOESI protocol, Xeon Phi presents the ownership *tag directory (TD)*. The possible limitation is given by the fact that there is a missing O state in the MESI, i.e. the MOESI avoids the need to write back the modified data. The ownership tag directory can be found in many other multiprocessor systems. The main purpose of the TD of the Xeon Phi is to emulate the missing O (owner) state in the MESI protocol. The MOESI protocol avoids the need to write back the modified data to the main memory before sharing it. The MOESI protocol is described in Table 5.

By adding the functionality of the TD to the MESI protocol, it is possible to emulate O state in the MOESI protocol without the need of redesigning blocks of cache. If the TD is faster than the memory, we get the promised performance boost compared to the MESI protocol. The tag directory implements the GOLS3 protocol. The GOLS3 protocol state diagram is shown in Figure 10 and the function of each state is described in Table 6. The modified coherence diagram for the emulated MOESI protocol is depicted in Figure 11.

Cache state	State definition
M (Modified)	The cache has only valid copy of the cache line and made some changes to the copy
O (Owned)	The cache is one of many with a valid copy of the cache line and has exclusive right to modify it.
E (Exlusive)	The cache has the only copy of the cache line and the cache line is unmodified
S (Shared)	The cache line is shared and consistent between cores. Is not consistent with memory (opposite to MESI). It has no right to make changes to the cache line.
I (Invalid)	The cache line is not valid. Must be fetched from memory.

Table 5: The MOESI protocol

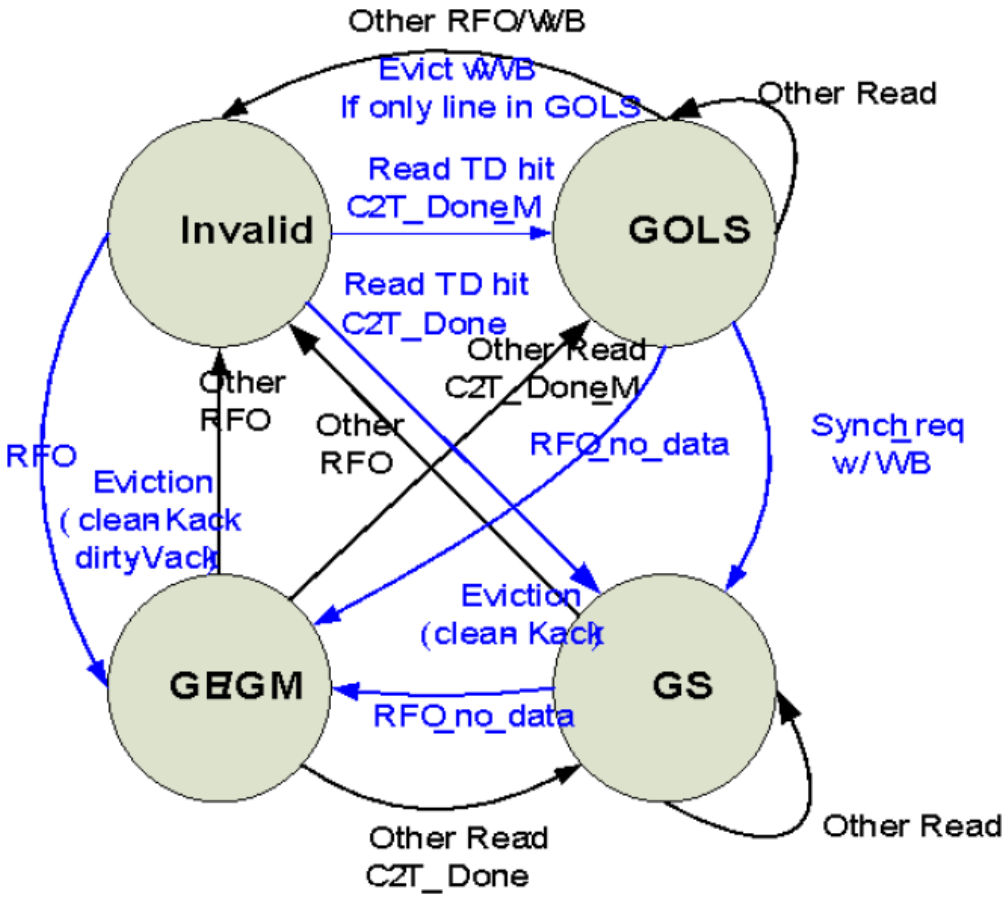


Figure 10: The GOLS protocol in the TD (source [8])

TD State	State definition
GOLS (Globally owned, locally shared)	Cacheline is present in one or more cores, but is not consistent with memory.
GS (Globally shared)	Cacheline is present in one or more cores and consistent with memory.
GE/GM (Globally exclusive/modified)	Cacheline is owned by one and only one core and may or may not be consistent with memory. The TD does not know whether the core has actually modified the line.
GI (Globally invalid)	Cacheline is not present in any core.

Table 6: GOLS protocols states

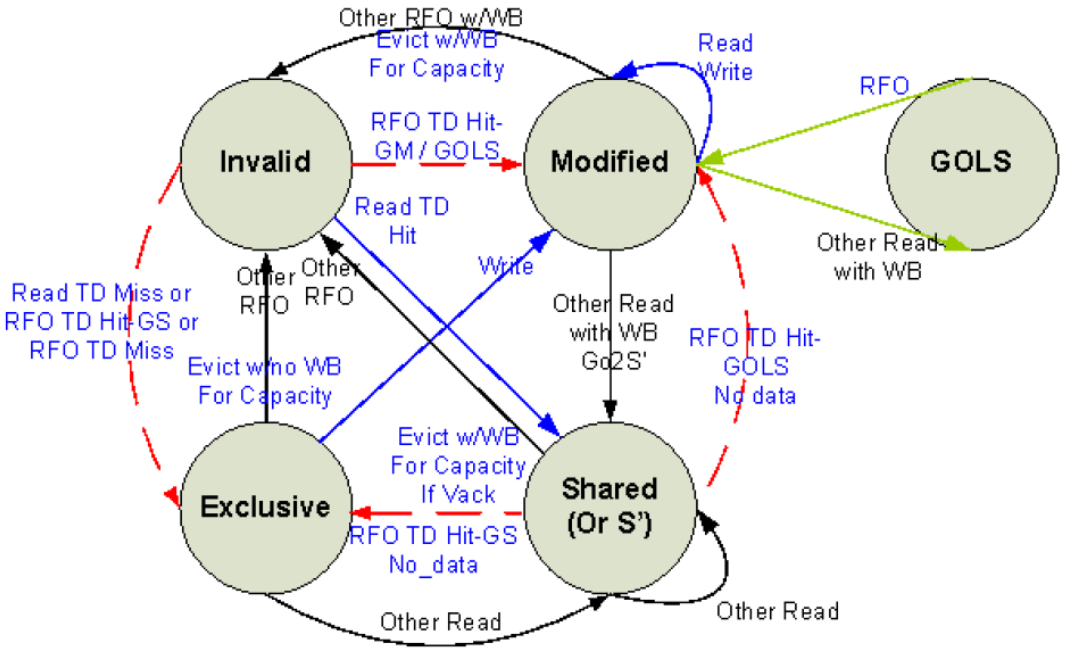


Figure 11: MOESI protocol emulation (source [8])

The tag directory (TD) is distributed among the cores in 64 smaller *distributed tag directories* (DTDs). Each DTD stores the global coherence state for its assigned cache lines. It contains an address, the GOLS3 state and the ID of the owner of the cache-line. The cache-lines are uniformly distributed among the tag directories. That provides a uniform traffic characteristic on the ring.

Information about the cache-line state is associated with each cache-line and it is stored in individual cores. The cache-lines state uses the standard MESI protocol. If a cache-line is in modified state, the core sends a request to the DTDs and gets back the GOLS3 state of the cache-line.

Page Tables and TLBs

There are two types of memory in the Xeon Phi – uncacheable (UC) and write-back (WB). The L1 data TLB supports three types of page sizes – 4KB, 64KB and 2MB. It can address 32-bit physical addresses or 36-bit physical addresses extension (PAE) in 32-bit mode or 40-bit addresses in 64-bit mode. The execute disable (NX) bit is supported, in contrast to the global page (GP) bit which is not supported.

If the L1 cache and the L2 cache miss happens, a 4-level page table walk is performed. The instruction INVLPG has the same behavior. There are no restrictions for mixing the page sizes in a single address block (2 MB), but they all have to be consistent. The Table 7 shows characteristics of TLBs located on the Xeon Phi.

As we can see, all TLBs are 4-way. Instruction cache does not support large pages. Entries in the TLB can be shared among threads with the same values of registers of the TLB (e.g. CR3, CR0.PG, CR4.PAE, CR4.PSE, and EFER.LMA).

Type	Size	Entries	Maps
L1 Data TLB	4KB	64	256KB
	64KB	32	2MB
	2MB	8	16MB
L1 Instruction TLB	4KB	32	128KB
L2 TLB	4KB, 64KB, 2MB	64	128MB

Table 7: TLB's characteristics

Memory Controllers

There are 8 on-die GDDR5 memory controllers. Each controller can operate two 32-bit channels, where a channel can deliver with the rate 5.5 GT/s. The controllers directly interface with the ring. They are responsible for reading and writing the data to the GDDR memory by translating the memory reads and writes to commands. It also supports the ECC data integrity feature.

2.1.3. Interconnect

The Interconnect is implemented as a bidirectional ring. It contains three independent rings in each direction. The largest - 64 bytes wide ring is the data block ring. The address ring is much smaller and it is used to send read/write commands and memory addresses. The acknowledgement ring is the smallest and also the fastest one. It is used to send the flow control and the coherence messages. The details are shown in Figure 12.

Whenever the L2 miss appears on a core, the core generates a request on the address ring and queries the tag directory. If the requested data is found in the L2 cache, a forwarding request is sent to that cache over the address ring and the request block is sent back through the data block ring. If the address is not found in any tag directory, the appropriate tag directory requests the memory from the memory controllers. The memory controller returns the data back to the core over the data ring. One data transfer and two address transfers are needed per a request. Therefore, the less expensive rings – address and acknowledgement rings - are doubled to increase the bandwidth requirements caused by many requests on the ring.

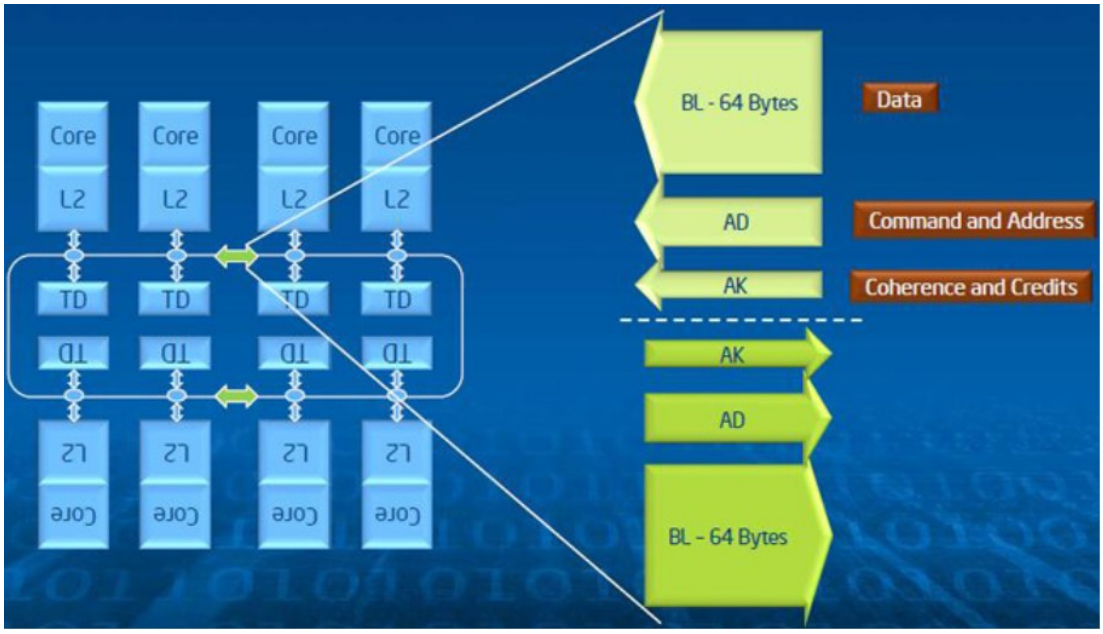


Figure 12: The interconnect (source [11])

2.2. Software Architecture

The Xeon Phi is implemented as a tightly integrated collection of processor cores on the PCI Express add-in card. The card meets PCI Express specification for the PCI Express endpoint. It implies that there are three address spaces (configuration, memory, I/O).

Each card represents Symmetric Multiprocessing domain (SMP) that is loosely coupled to the computing domain represented by the host platform. There are lots of APIs to support many High-Performance Computing (HPC) applications. There are also TCP/IP, MPI, OpenCL APIs and some other Intel interfaces, which create abstraction layers.

The communication layer between the host and the card is called the SCIF. They altogether create Manycore Platform Software Stack (MPSS). The MPSS is the collective name for all software on the host and the card that supports the coprocessor.

Figure 13 shows relationships between the APIs and other components. The left side of the picture shows the host stack that runs on standard Linux kernel. The right side shows a similar stack that belongs to the coprocessor and runs on top of the Linux based kernel with the Xeon Phi adjustments.

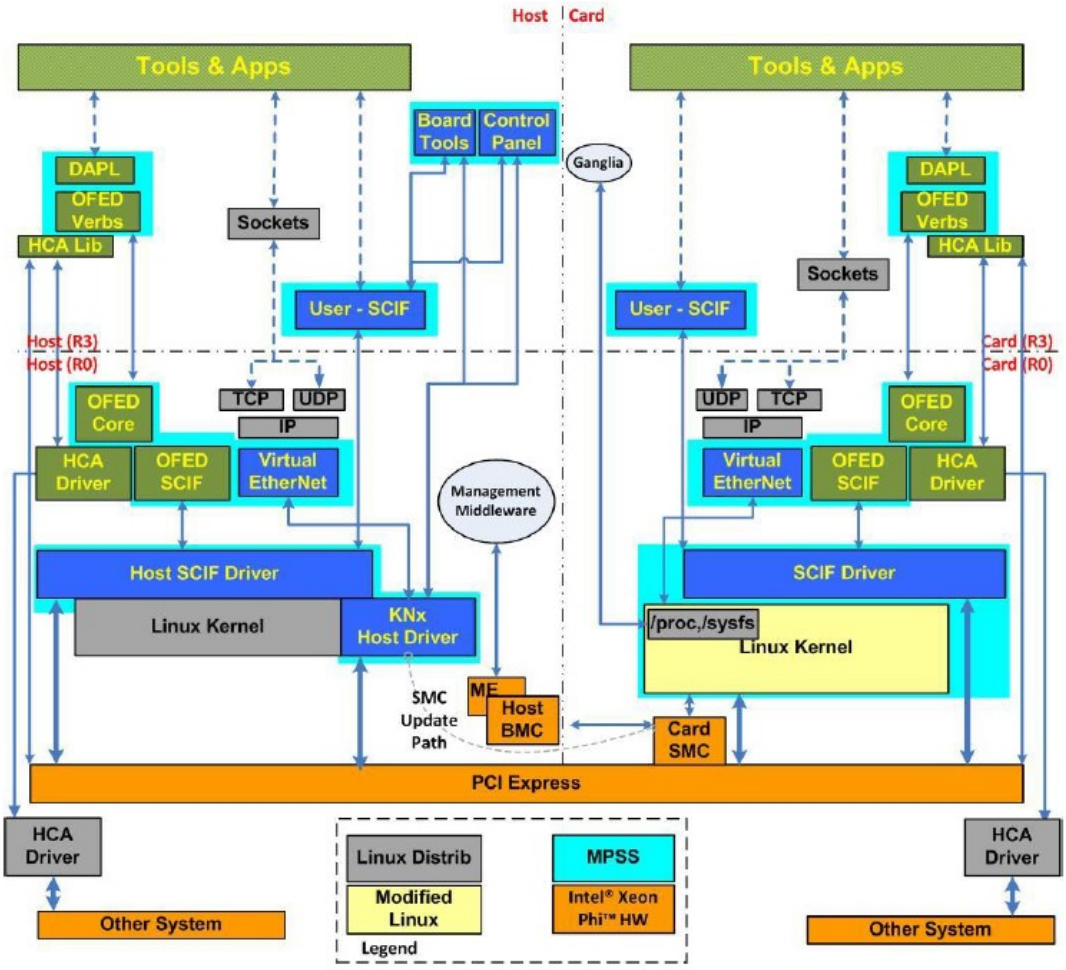


Figure 13: The Xeon Phi software architecture (source [8])

The software architecture supports three programming models – offload, symmetric and native. In the offload model, the main application is launched on the host processor. The offload process is controlled by some compiler directives and it is fully in the scope of activity of the compiler. The symmetric model is composed of multiple processes, where the computation and the communication need to be done explicitly by using some standard mechanism e.g. the MPI. The last native model is the variant, where the application is running exclusively on the Xeon Phi coprocessor.

Unlike standard computers, the Xeon Phi does not contain a permanent file system (FS) storage. Its FS is maintained in RAM and it can be remotely mounted e.g. through NFS.

2.2.1. MPSS

The high level representation of the MPSS is pictured in Figure 14. Although the software stacks of the MPSS contain similar components, they are not binary compatible, because they both run on different platforms. Logically, the same thing

holds for applications – the native Xeon Phi applications are not binary compatible with any other applications.

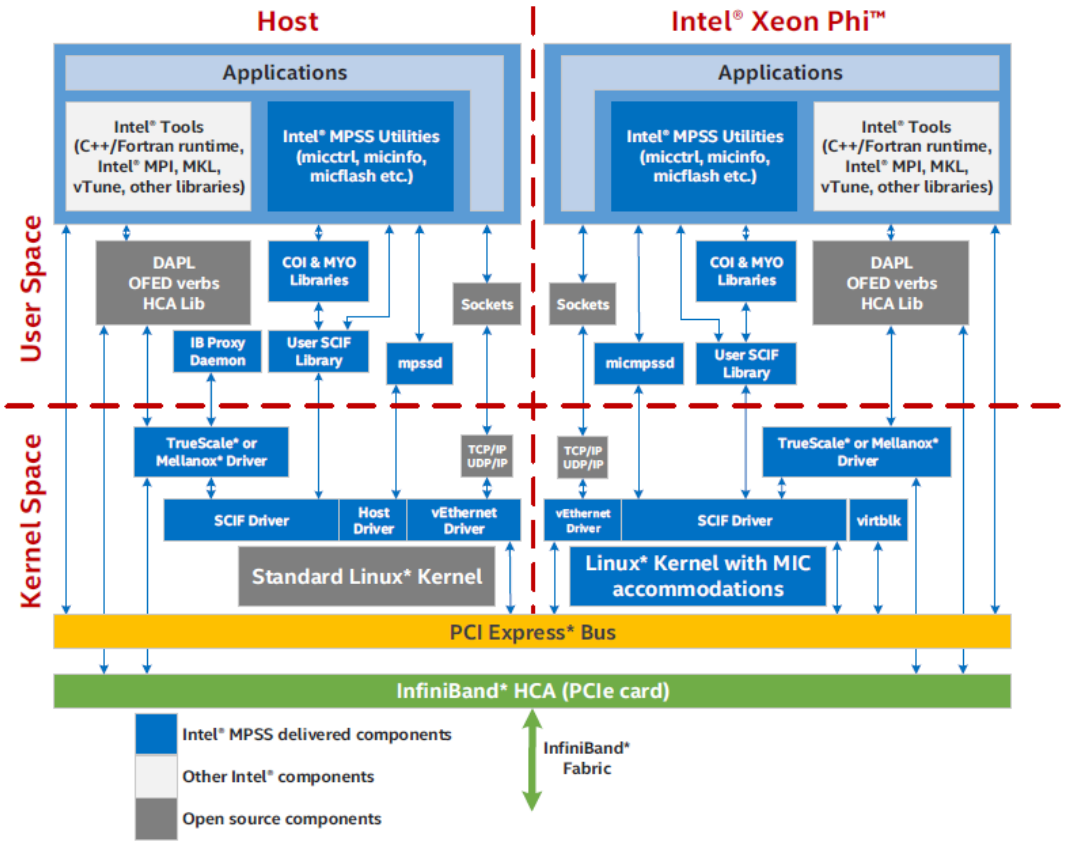


Figure 14: The MPSS (source [8])

The kernel used in the coprocessor is a standard Linux kernel with a few adjustments due to the special architecture. The kernel and an initial FS image are stored in the MPSS part of the host. The host driver injects the Linux kernel image and the kernel command line into the memory of the coprocessor to begin execution. The kernel command line and the FS are constructed based on configuration files. The files can be configured by certain host configuration files in the MPSS.

All Linux utilities are provided by BusyBox [12]. Busybox is a program that provides several stripped-down UNIX tools in a single application. It was originally developed for operating systems with limited resources.

There is a virtual ethernet transport between the host and the coprocessor implemented by virtual ethernet drivers. It supports the TCP, UDP, IP stack and tools such as the SSH, SCP etc.

As mentioned previously, the SCIF is a low-level communication layer between the host and the coprocessors. It also provides a communication between multiple coprocessors within the platform. It exposes DMA capability for transfers and

an ability to map memory from the coprocessor to the host and from the host to the coprocessor. The SCIF is built on top of the PCI Express, which results in the advantage of an inherent reliability and no need to error check any data packages compared to the virtual ethernet driver.

2.2.2. Overlay System

There are prepared system administration tools for the Xeon Phi on the host. They are used to control the coprocessor. Alternatively, the coprocessor can be controlled through configuration files. They are located in */etc/sysconfig/mic/* on the host. There is a configuration file for all coprocessors installed on the system – *default.conf* and the configuration files for each installed coprocessor – *mic<n>.conf* (where *<n>* is the number of the coprocessor). There are also the configuration files of the installed software– *conf.d/**. Each kernel module is configured by using the file */etc/modprobe.d/mic.conf*. Every change to the file requires a restart of the coprocessor for changes to take the effect.

The root file system for coprocessor is constructed on the host. The result is compressed into a cpio file. On boot, the content of the file is unpacked and loaded into the RAM disk. The file system image cannot be directly modified, but instead of that one or more file hierarchies overlay corresponding files in the file system image during the boot. It means that each file in the hierarchy replaces the corresponding file in the base file system or creates it, if it does not exist in the base file system. The hierarchies are called *overlay sets* or *overlays*. The used overlays are described in the configuration files. For example, if the file *foo* is created in */var/mpss/common/etc/foo*, it overlays */etc/foo* in the file system of each coprocessor.

The overlay process begins with the *Base* file system. Its configuration specifies the file system to be used. The common overlay can include files that overlay the file system of all coprocessors. The common overlay is rooted at */var/mpss/common* by default.

There exists a per-coprocessor overlay that enables overlaying of files for the coprocessor. The overlay includes some files by default (see [13] for more information).

3. Testing

The purpose of the tests is to find the missing details or wrong information about the architecture of the Xeon Phi. Sometimes, the details are left out intentionally to protect the secret of a company, and sometimes they are missing by mistake.

The chapter starts with the description of preparation of the testing environment. It continues with the description of the testing methodology. There are special methods to ensure that the results of the tests are reliable. We describe the architecture of the Xeon Phi based on the results. The tests are focused on measuring the memory latency for shared and unique access.

3.1. Preparation

The testing environment needs to be prepared at first. Even though the Xeon Phi uses a Linux kernel, it has its specifics. Some of them cannot be found in the documentation and they are usually discovered on the fly. In this chapter, we describe the specifics that we found.

Compiling a Kernel Module

Building a kernel module for Xeon Phi is not so different from building a kernel module for another platform [14]. The infrastructure for building a kernel and managing all compilation options is known as the Kernel Build System – *kbuild*.

The kernel was built on the host, because there are no building utilities on our Xeon Phi. This leads to another problem – we have to cross compile the kernel module.

Firstly, we take the source codes of our kernel. It can be downloaded or accessed on the Xeon Phi through the NFS. Secondly, we use the compiler that is able to create binary files for the Xeon Phi. There exists a version of GCC especially prepared to produce binaries for the Xeon Phi. All required libraries together with the compiler need to be given to the kbuild, e.g. through the *PATH* variable. The last step is to start a build with the corresponding architecture.

Loading a Kernel Module

The load of a kernel module can be done in two ways. The first possibility is to load it while the coprocessor is booting. This implies that the kernel module has to be defined in overlays. The disadvantage of this solution is that the coprocessor must be rebooted with every change to the kernel module. An easier way to load a kernel

module is with the standard utilities *insmod* and *rmmod*. To call the utilities, we need to gain root privileges on the coprocessor.

Root Privileges on the Xeon Phi

On the standard Linux, the root privileges are gained if there is an appropriate entry in */etc/sudoers* file. We cannot modify the file directly, because the changes last only until next reboot. The overlay must be used to preserve the changes. However, if the file is added into an overlay definition, nothing happens. There is nothing about this special behavior in the documentation [13]. Nevertheless, there is a note that some files are included by default into a per-processor overlay. Except for *sudoers*, almost all special files from */etc* directory are included. It seems that the files from */etc* are either included by default into overlay or they cannot be included at all.

Luckily, there is an entry in default *sudoers* for additional include directory (*sudoers.d)*. All files from this directory are included into *sudoers*. Therefore, the final solution to add a user into sudoers is to create a file with the desired rule and force the overlay to copy the file into */etc/sudoers.d*/ directory.

3.2.Methodology

At the beginning, we decide how to measure our code properly. The first option is to measure a time from the user space. The second option is to measure it from the kernel mode. Because we need to access some kernel structures and some kernel functions, we must create a kernel module anyway. That is why we have decided to measure our code from the kernel mode by reading a timestamp counter. The results given by this method should be more accurate.

RDTSC and CPUID

All Intel CPUs have a monitoring counter to keep track of events that are happening inside a processor. One of the counters is the timestamp counter, which counts a number of cycles in the processor. This counter can be accessed by *RDTSC* or *RDTSCP* assembly instructions. The *RDTSC* was introduced by the Pentium processor. The *RDTSCP* was introduced later and it is not supported by all processors. Especially, the Xeon Phi does not support it, so we must use the *RDTSC* version.

Both instructions read timestamp counter (a 64-bit *MSR*) into the *EDX:EAX* registers. The *EDX* contains the high-order 32 bits of the *MSR* and the *EAX* the low-order 32 bits. The *RDTSC* is not a serializing instruction. That means that it does

not wait until previous instructions are executed before reading the counter. That is the main difference from the waiting *RDTSCP*. On the other hand, the subsequent instructions might start an execution before the read operation begins. Both instructions have this in common. In order to measure intended instructions, we need to include a fence before and after the *RDTSC* instruction.

The Intel architectures define several serializing instructions. These instructions force the processor to complete all modifications of flags, registers and memory, produced by previous instructions. They also force all buffered writes to memory to drain before the next instruction is fetched. The implication is that nothing can pass the serializing instruction and the serializing instruction cannot pass any other instruction.

The *CPUID* is one of the instructions used for serialization. Its main purpose is to return an identification of the processor. It guarantees that any modifications to flags, registers and memory for all previous instructions are completed before it is fetched and executed.

The right combination of the instructions above ensures that we are able to measure only the piece of code that we need, and no other instructions. At the beginning, we place the *CPUID* instruction before the *RDTSC*. We know that the *CPUID* serializes instructions. It is obvious that all instructions are executed in advance. Also, the *CPUID* cannot pass through the next instruction – the *RDTSC*. At the end of the code, the *CPUID* is placed right before the *RDTSC* instruction. The *CPUID* ensures that the instructions after do not start execution before the *RDTSC* reads the counter. One disadvantage is that we measure also the latency of the *CPUID*. If we could use the *RDTSCP,* we would easily solve the problem with measuring the latency of the *CPUID* – if the *CPUID* is placed at the end before the *RDTSCP,* it is able to serialize instructions before.

The described process for measuring is general and can be used for any out-of-order processor with instruction pipelining. The measurement of the Xeon Phi is easier, because all the cores of the Xeon Phi are in-order, so the coprocessor cannot internally reorder instructions. However, there is a support for pipelining in the coprocessor; therefore the serializing instruction must be used. It forces any instructions in any pipeline stage to finish before the *RDTSC* instruction starts to measure.

Disabling Kernel Preemption

There are other factors that might ruin our measurements. The OS of the coprocessor is preemptive. If its scheduler decides, it can temporarily interrupt a running thread and let another thread to run instead. In our test scenario, is it unlikely to happen, because we run only few threads. The only thing that might ruin the measurement are interrupts.

Our goal is to exclusively own the CPU and then run our code. We call functions `preempt_disable()` for disabling preemption and `raw_local_irq_save(flags)` for disabling hard interrupts in the code. To return both thing back to normal, we use functions `raw_local_irq_restore(flags)` and `preempt_enable()`.

3.3. Testing Architecture Layout

As we stated before, the cores and the memory controllers are placed on the bidirectional ring. We work on the assumption that the cores are uniformly distributed among the ring and the memory controllers are placed evenly between them, as shown in Figure 8. We can also see that the memory controllers are paired.

We test whether the access from some CPUs to some controllers takes longer than to the other controllers. It would imply that the system has the NUMA architecture. We also measure the impact of accessing to the DTD from different cores.

3.3.1. Memory Controllers

First, we test the latency of the access from a core to a memory controller. The architecture picture shows that the cores are uniformly distributed among the bidirectional ring. It also shows that the memory controllers are paired, which leaves us with 4 pairs of memory controllers (there are 8 memory controllers). It seems reasonable that those groups are placed uniformly between the individual cores. Because the Xeon Phi has around 60 cores, there must be around 15 cores between adjacent memory controllers. If we take opposite memory controllers and the closest core to the first controller, there are around 30 cores between the core and the second controller. This should be the longest path between a core and a memory controller, because the ring is bidirectional, and it uses the shortest path between every two elements on the ring. We think that there is a difference between the access time to the memory controller from the nearest core and from the furthest core.

Physical Addresses to Memory Controllers Mapping

At first, we need to know how the addresses are mapped to the memory controllers. The information can be read from the performance counters, which lie inside every memory controller. Unfortunately, we did not find any public document describing where the performance counters are placed. It seems that it is not a public piece of information. If we knew, where the performance counters were, we would try to read and write cache-lines one by one while looking into the performance counters.

It does not seem that the blocks of memory in each controller are smaller than a cache-line. We expect that each controller contains block of memory that corresponds to the size of the virtual pages. The 4KB block size seems reasonable. That makes 64 cache-lines in each block, because the size of a cache-line is 64B.

There is a proposed layout of memory on the Intel's forum [15], but it has not been validated by anyone. Physical memory of the coprocessor is divided into blocks of 62 cache lines. The blocks are interleaved around the 8 memory controllers with 2 channels per controller using a fixed repeating pattern. The left 2 cache-lines in each 4KB range are used for holding ECC data. The suggested layout is depicted in Figure 15.

There is also proposed mapping between the memory controllers and the physical memory addresses. The mapping is written in Table 8. We test the proposed mapping and validate characteristics of given results. The characteristics must correspond with the well-known information about the architecture of the Xeon Phi.

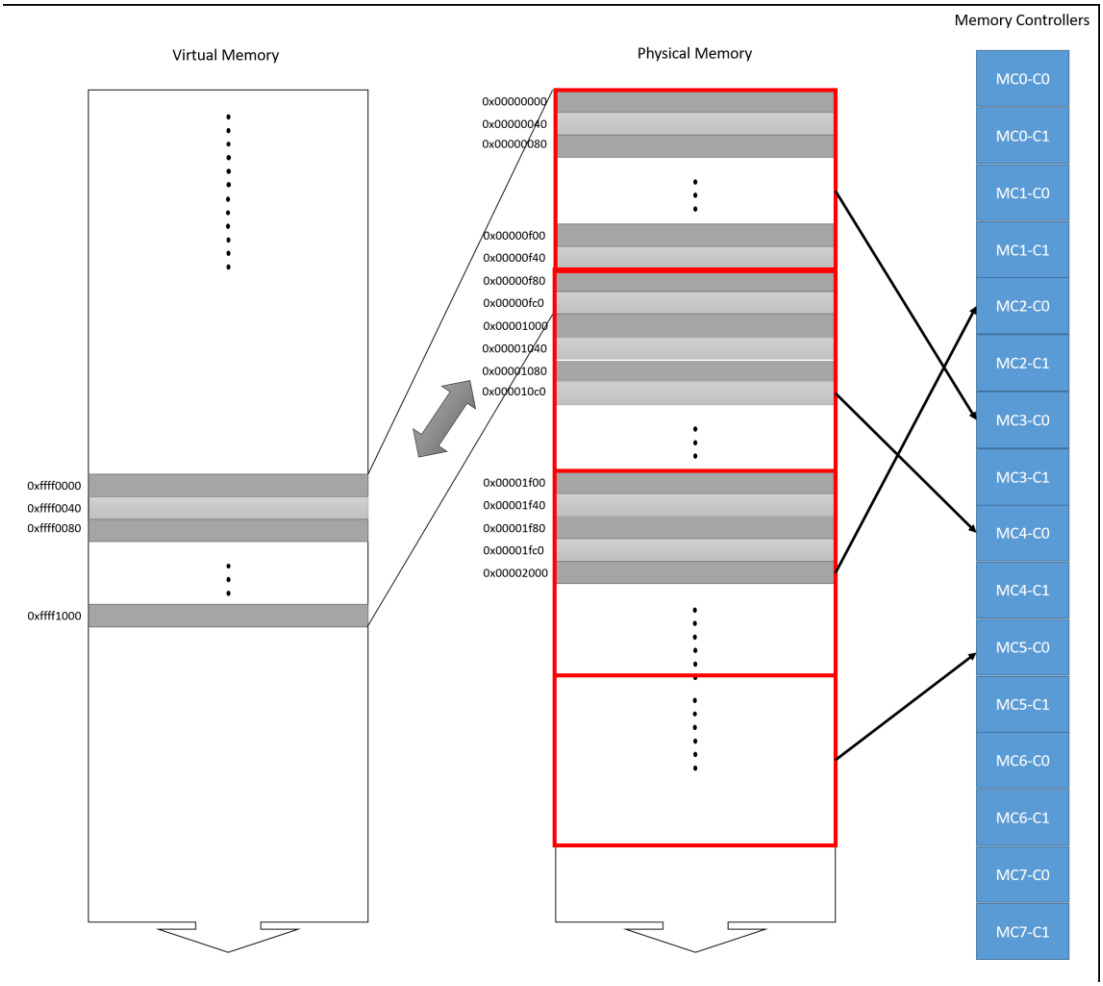


Figure 15: Memory controller mapping

Select = (Physical Address / 62 / 64) & 0xF		
Select	Memory Controller	Channel
0	3	0
1	4	0
2	2	0
3	5	0
4	1	0
5	6	0
6	0	0
7	7	0
8	3	1
9	4	1
A	2	1
B	5	1
C	1	1
D	6	1
E	0	1
F	7	1

Table 8: Physical address to memory controller mapping

Implementation

For measuring latencies of a memory controller, we must access only blocks that belong to a single memory controller. We use a technique called a pointer-chasing. We create a big array of unsigned integers. Each of them is the index of the next element in the array. It basically creates a chain of the pointers. Our program traverse the array using a read element as the index of the next element. Because we use the read value as the index, a prefetcher cannot prefetch the code in advance. That allows us to see the latency of reading from the memory.

The Algorithm 1 shows the traversal code. The array in the example is called *A*, and the index of the array is called *i.* We do not use this exact code in our testing program. We manually unroll the cycle instead. Not optimized cycles are replaced with comparisons and jumps in the assembly code and we want to avoid them. We need to measure only the access to the memory.

```
i = 0;
while(i < A.size()) {
        i = A[i];
}
```

Algorithm 1: The pointer-chasing algorithm

We create the array with the chain of pointers that jump only on memory blocks stored in each memory controller. The cache-line is 64B long, each block has 62 of them and the blocks are repeatedly distributed among memory controllers. If we want to jump on the first cache-line in a memory block of each memory controller, we must jump over $64B \times 62 \times 8$. To skip the channels of the memory controllers, multiply the given number by 2 (there are 2 channels per a memory controller).

We allocate 2^{10} of 4KB pages, which makes together 4MB of a continuous space. We cast it into an array of *uint64_t*. The length of the pointer chain between blocks is the size of the array divided by the length of a jump. That gives us the pointer chain with at least 64 jumps ($\frac{4MB}{(64\times62\times16)} \geq \frac{4MB}{2^{16}} = 64$).

We run the test multiple times. In case of multiple runs, we must take caches into consideration. If the test runs for the first time, the values are loaded from a main memory – a memory controller. When the values are loaded, cache-lines are stored into caches. We get bad results every other run, if we do not erase the values from the caches. Caches of the Xeon Phi are using a modification of LRU algorithm to change cache-lines. Therefore, we must replace the old values with some new values. We

create an array with the size of L2 cache – 512KB. To fill the caches with the new values from the array, we simply iterate through the array few times.

One of the problems that might arise, during the creation of the function that fills up the cache, are optimizations on some compilers. The compilers are able to optimize away useless things. A function for filling the cache with some dummy values is a great example. A basic implementation goes through an array and does nothing. To prevent the optimizations, either the program must be compiled without the optimization enabled or the program must do some work with the cache-lines e.g. reading/writing values.

Kernel Specific Functions

A memory in kernel can be allocated in two ways. One of them is to use `kmalloc()` and `kfree()`. These functions are intended for allocation of smaller chunks of the memory. They are similar to user space `malloc()` and `free()`. There is a second pair of functions – `__get_free_pages()` and `free_pages()` that can be used for allocation and deallocation of whole pages. These pages translate into contiguous physical pages.

The memory allocated in kernel is un-swappable by default. It means that nobody can swap the memory out. If we used a classic user space allocator, which returns a swappable memory, the memory could be relocated and mapped to totally different physical page. That might imply a different memory controller associated with the new physical page. However, there are possibilities how to pin a memory even in the user space.

To run a single thread on a core and to forbid scheduling of the thread to another core, we need to pin threads to cores. There exists the function `kthread_bind()` for this purpose. One of the parameters is a *task_struct* pointer that holds the information about the thread. The second parameter is a CPU number to which we want to bind. The number is not exactly a core number, but it is a number of the thread. For our Xeon Phi with 61 cores, each having 4 threads, the possible CPU numbers are from 0 to 243. In order to bind a thread to a CPU, the thread must be stopped. That is the difference between the function `set_cpu_allowed()`, which is used for the implementation of `kthread_bind()`, and function `kthread_bind()`.

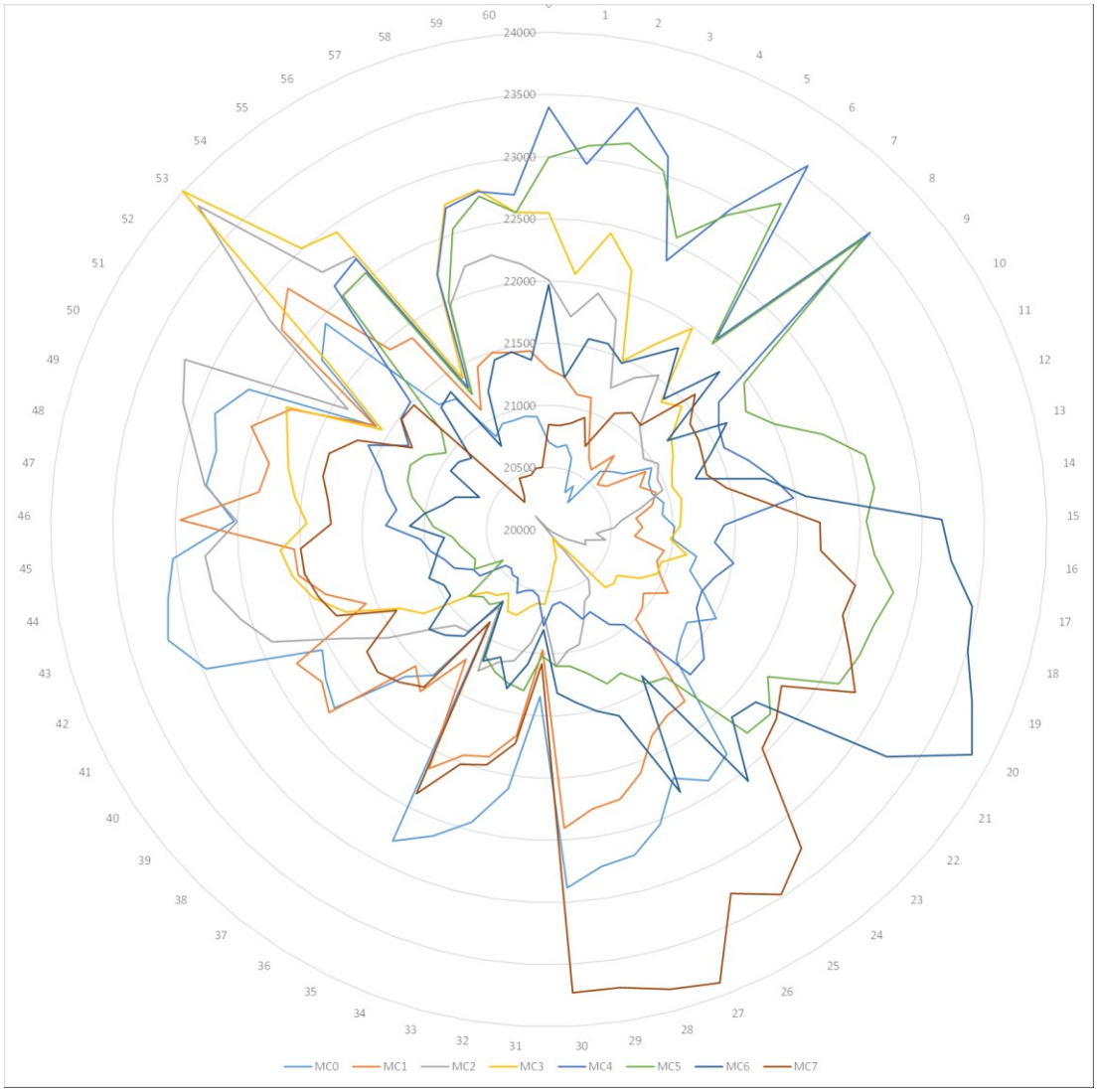
Another possibility to bind a thread to the CPU is offered by the OpenMP. However, the framework cannot be used in kernel space.

Last special kernel function is virt_to_phys(), which is used for translation of virtual addresses into physical addresses. It works exactly as anyone would have expected – it gets a virtual address and returns the appropriate physical address.

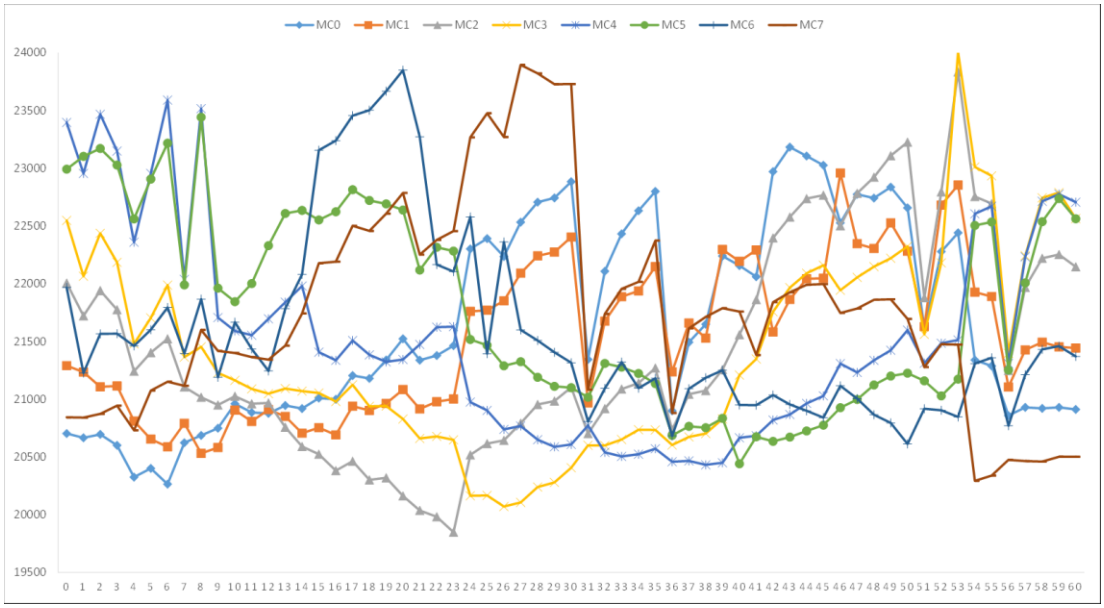
Results

We measure the latency between each logical core (a physical core with its threads) and each memory controller with its channels. One measurement cycle does 64 jumps in the memory on a single memory controller. After the measurement cycle, caches are flushed and we measure again. The cycle is repeated 1024 times. We calculate an average value from all measured results. The higher number of repeats was not able to achieve, because the kernel starts timing out whenever the loading of a kernel module takes a long time. This process is repeated for each logical core (244 of them) and each memory controller with its two channels (16 of them). That gives us together 244×16 combinations.

The results show the latency of accessing the memory without any synchronization. They are shown in Graph 1. Individual cores are placed around the circle. Each color represents the latency of a memory controller. The closer the value is to the center of the graph, the smaller the latency gets. The second view on the same results is depicted in Graph 2.



Graph 1: Read latency from cores to memory controllers



Graph 2: Read latency from cores to memory controllers

The Latencies of Memory Controllers

The values are approximately between 20000 and 24000 cycles per 64 memory accesses. It gives us the values from 310 to 375 cycles per a memory access. The results correspond with expected values. The read latency of main memory measured in [3] was around 305 cycles per a memory access. There is a difference in order of magnitude compared to reading from the L2 cache, which takes 11 cycles. Also, there is 65 cycles difference between the fastest and the slowest reading from the memory.

We assume that the fastest read is when the core is next to the controller. The slowest read should happen when the core is placed in opposite to the memory controller. This behavior can be seen in Graph 1. The highest values are always opposite to the lowest ones. Also, the sum of opposite values is always the same. In the other graphs, we see that the change between adjacent values is almost linear. There are around about 30 cores in one half of the ring. With the difference around 60 cycles, it gives us the latency 2 clock cycles per a core on the shortest path between the memory controller and the core.

Xeon Phi Architecture Layout Results

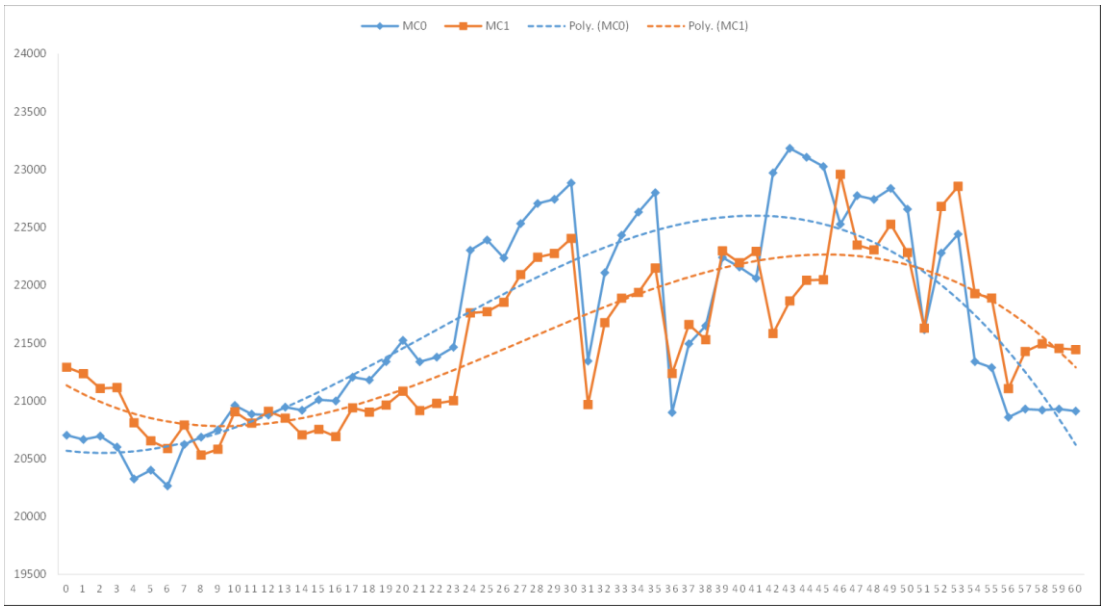
As we already know from the architecture pictures, the memory controllers are coupled. Therefore, the access latencies of those controllers must be similar. That holds for the memory controllers with indices (0 + 1), (2 + 3), (4 + 5) and (6 + 7).

The latencies of the found memory controller pairs are depicted in Graph 3, Graph 4, Graph 5 and Graph 6. We include a polynomial trend line to see how the latencies behave as a complex. Every graph contains precisely two intersections of the trend lines. One of them is near the minimum and the other one is near the maximum. Naturally, the intersections correspond to the intersections that are between the lines of the latencies. They are also placed near the extreme values and they swap the relation of the values.

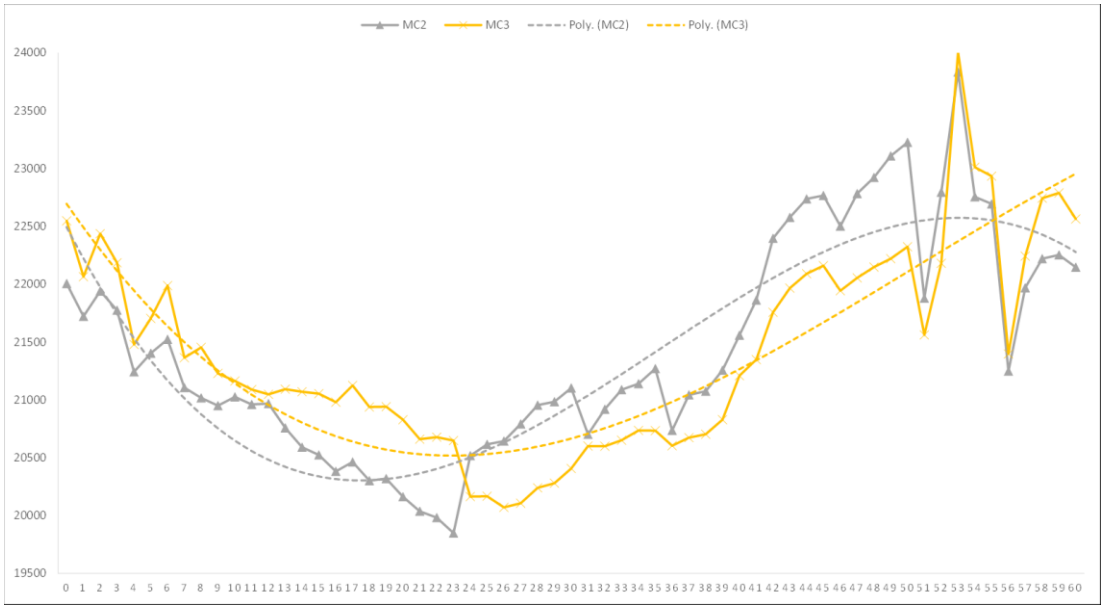
The extreme values split the graph into three areas. In the first area, most of the latencies of the controller A are higher than the majority of the latencies of the controller B. This relation inverts in the second area. The latencies of the controller B are higher than the latencies of the controller A. In the third area, the relation between the latencies inverts again, hence it is the same as in the first area. Most of the latencies of the controller A are higher than the majority of the latencies of the controller B.

This behavior corresponds with the layout of the memory controllers and the cores on the ring. If we get closer to a memory controller pair from the one side of the

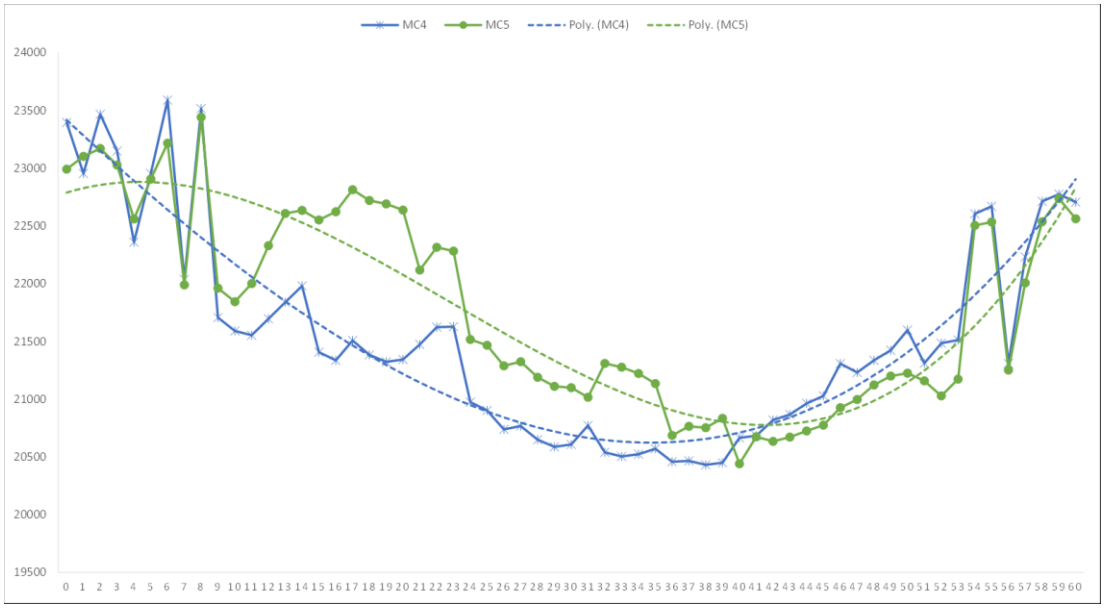
ring, the latencies fall down. Clearly, we are always closer to one of the memory controllers than to another from the pair. The second controller is hidden behind the first controller, so the path to it is always one step longer. It changes when we get to the other side of the memory controller pair. Now, the previously hidden controller is closer to the cores. The second change happens on the opposite side of the ring. As we stated before, the ring is bidirectional. Therefore, requests always take the shortest path to their destination. There is a place where the shortest path stops moving one direction and starts moving the other direction. For the requests of a memory controller, the place is right opposite to the memory controller.



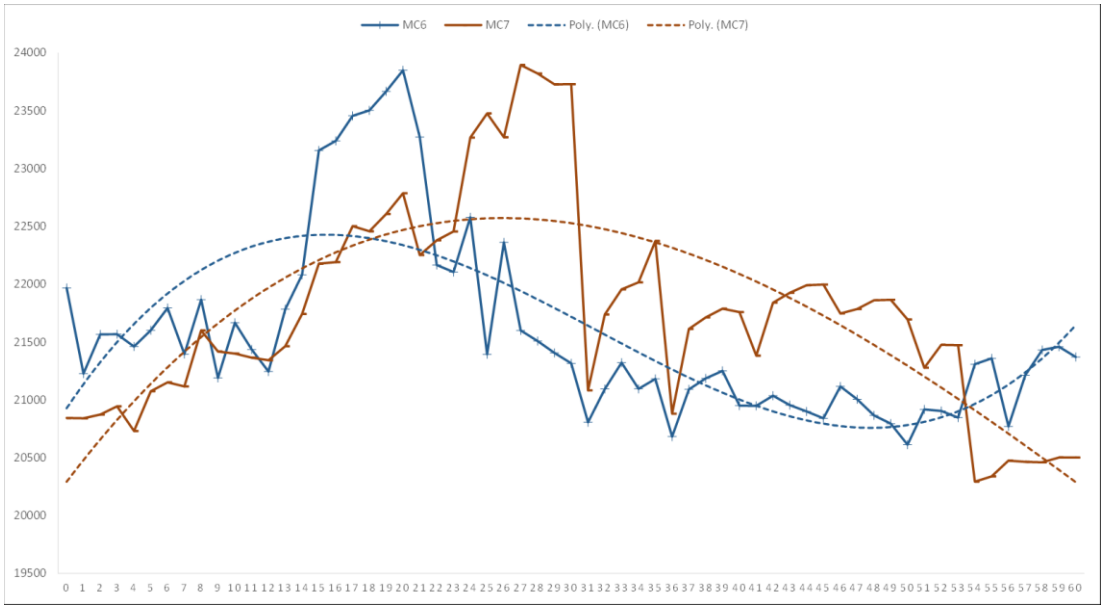
Graph 3: The latency of memory controllers no. 0 and no. 1



Graph 4: The latency of memory controllers no. 2 and no. 3



Graph 5: The latency of memory controllers no. 4 and no. 5



Graph 6: The latency of memory controllers no. 6 and no. 7

NUMA Effect

Base on the results, we propose the layout of the memory controllers and the cores on the ring. The cores that are the nearest to the intersection with the minimal latency value surround the memory controller. We create the layout based on this idea. The layout is shown in Figure 16.

Just to recap, the latency of accessing the memory controller is between 310 cycles (for the closest core) and 375 cycles (for the furthest core). That gives us 60 cycles difference. These 60 cycles are used to traverse a half of the ring, which contains 30 cores and one memory controller pair. If we want to create some NUMA nodes, we should create them around the memory controllers as shown in Figure 17. The NUMA

nodes are in the red ellipses. An average distance between the cores from the adjacent NUMA nodes is 30 clock cycles. If the fastest latency is around 310 clock cycles, then the NUMA factor is 1.1.

The NUMA factor is so small that it should have no effect on the majority of the applications. A high performance application can gain a small performance boost if it utilizes the proposed NUMA architecture. However, this NUMA factor holds only for unshared memory. If the memory is shared, the DTDs get involved and the latencies of the access to the memory are completely different.

Physical Address to Memory Controller Mapping Validation

The trend lines of the latencies of every memory controller are depicted in Graph 7. The similar graphs for the memory controller pairs are in Graph 8 and in Graph 9.

The graphs clearly show that the memory controllers are placed on the ring and their latencies correspond with the architecture of the Xeon Phi. Moreover, they validate our hypothesis that the memory blocks are built from 62 cache lines. If it was not true, the characteristics would not correspond to the circle layout of the ring inside the Xeon Phi. So, it is correct to say that the mapping in Table 8 is correct.

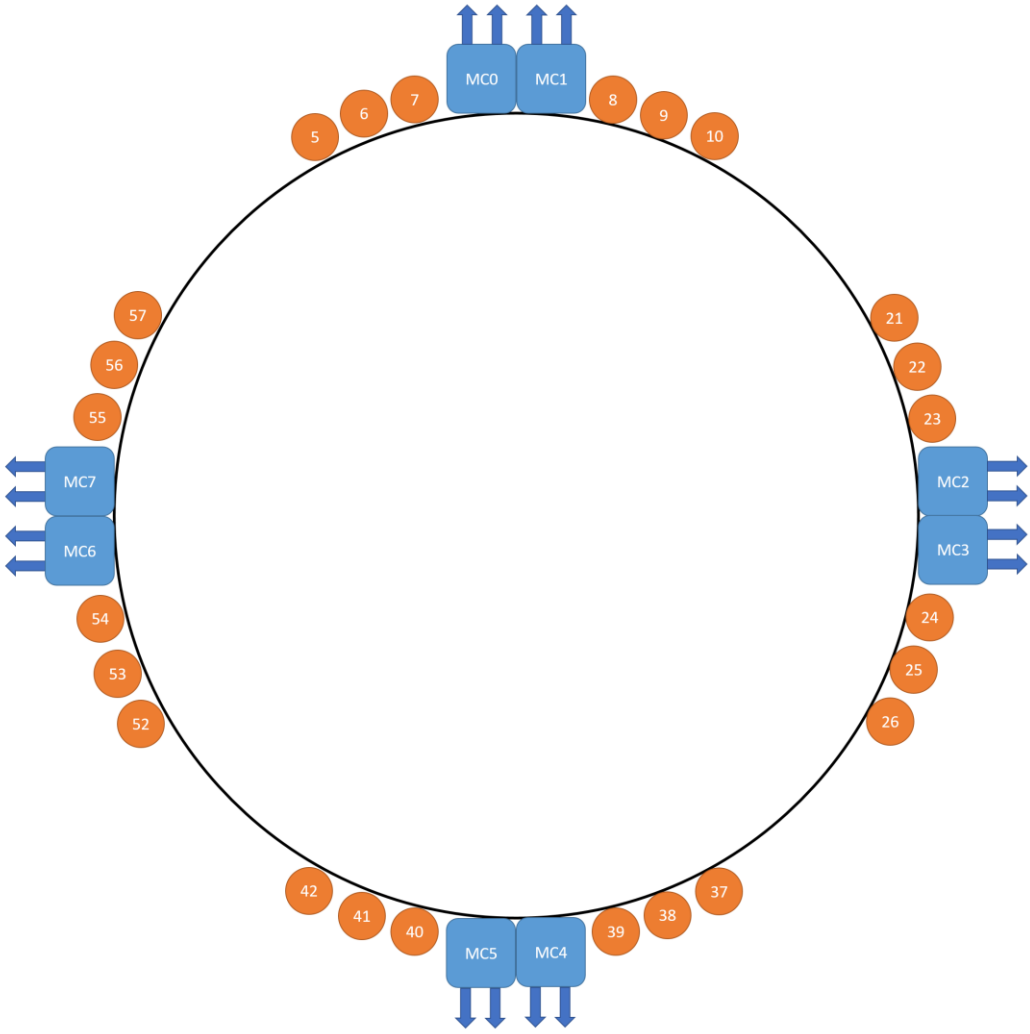


Figure 16: The layout of memory controllers and cores

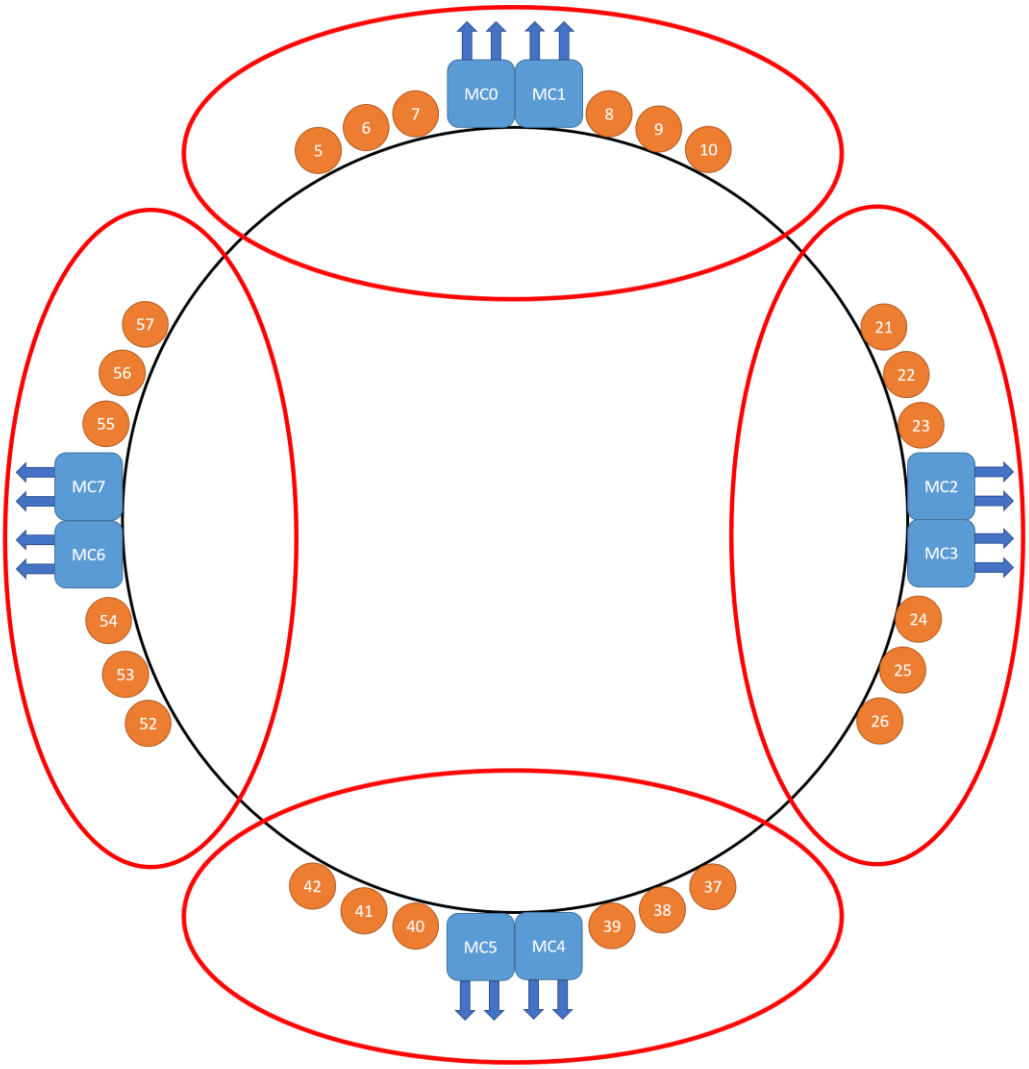
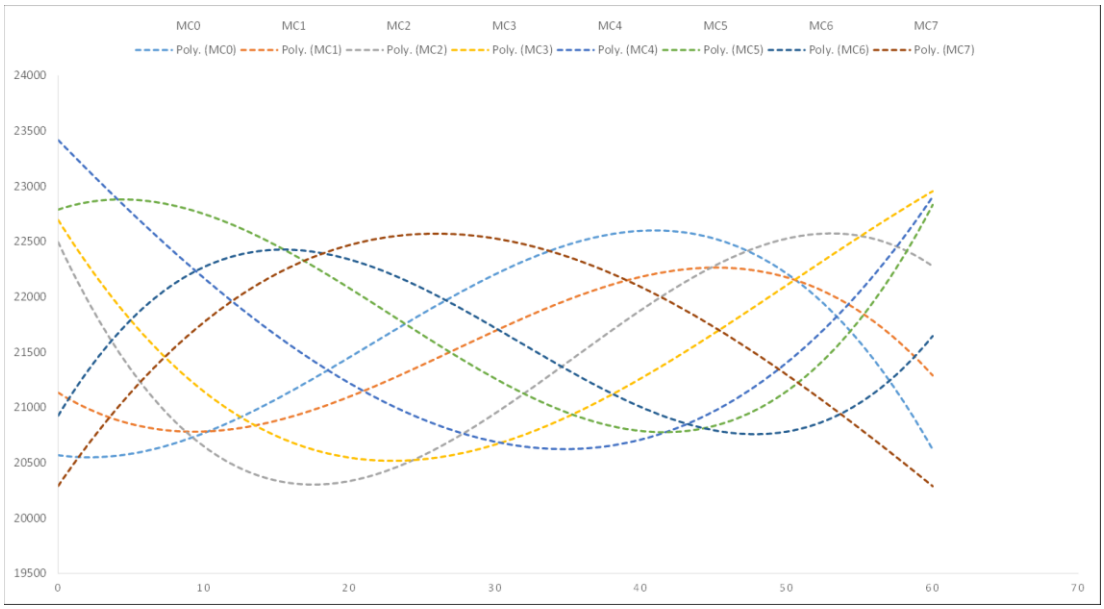
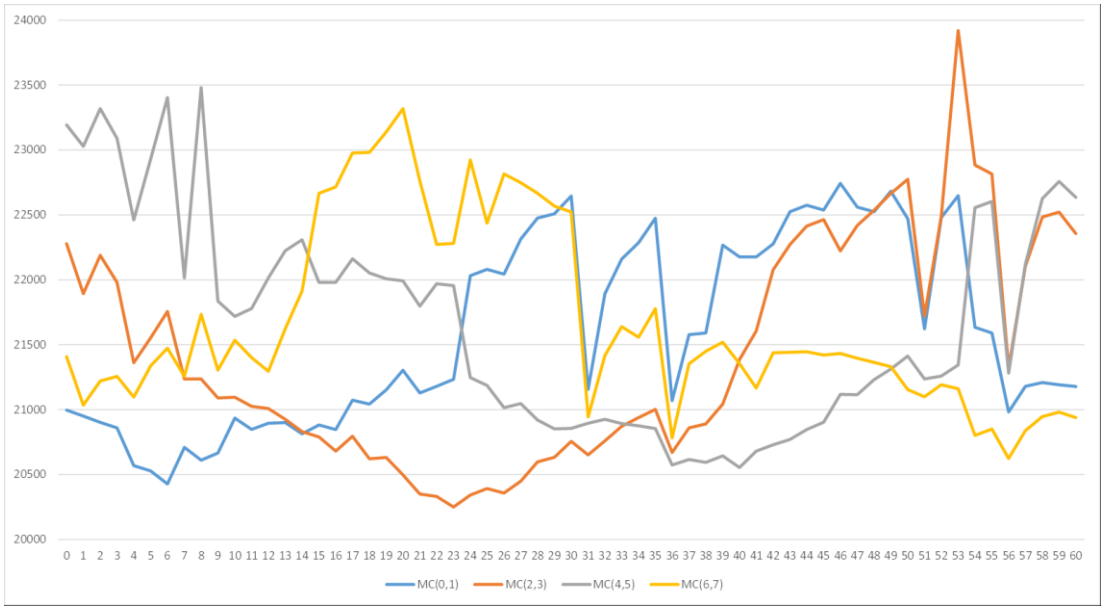


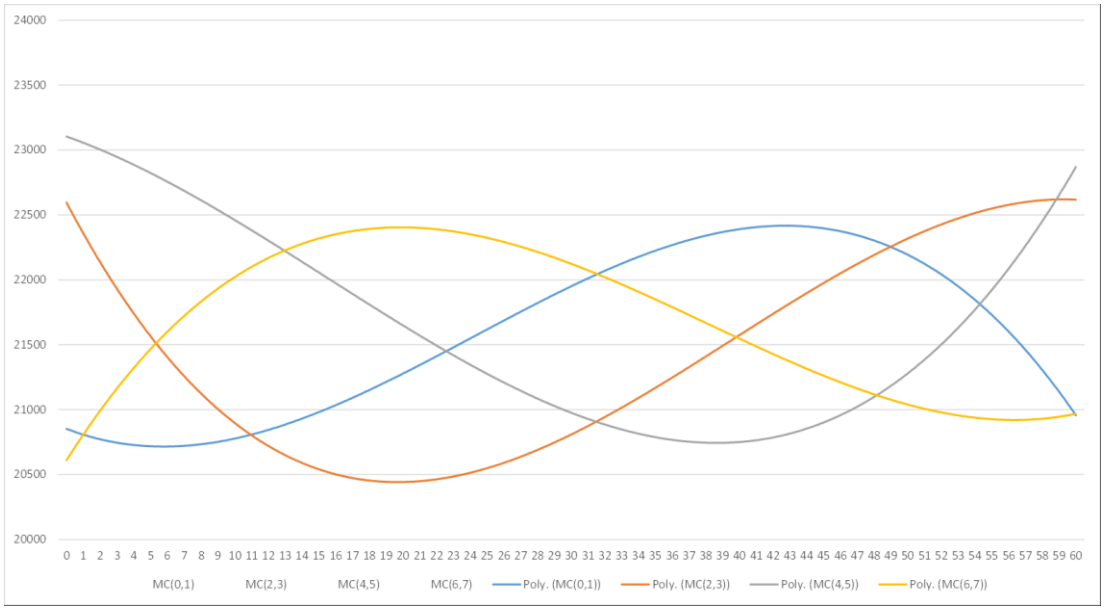
Figure 17: Proposed NUMA nodes



Graph 7: The trend lines of latencies of each memory controller



Graph 8: The latencies of memory controller pairs



Graph 9: The trend lines of the latencies of memory controller pairs

3.3.2. DTDs

The second thing that we test is the latencies of the DTDs (distributed tag directories). There are 64 tag directories (TD) around the ring. One record in the TD contains a cache-line address, a mask with valid CPUs and a state. It is hard to get information about the mapping between cache-lines and the TDs. There are no performance counters available for the TDs. One of the possibilities how to find a mapping is to test it.

Our application tests a cache-line ping-pong of different cache-lines between the two cores. Because the TDs are uniformly distributed among the ring, we expect different results for different TDs.

Implementation

We create a cache-line ping-pong for testing the latencies of the TDs. Our testing application has two threads. The first thread accesses a cache-line. Because it does not have the cache-line in its cache, it sends a load request to the TD owning the cache-line. As no core has the cache-line in its cache, the TD sends a request to the memory controller owning the address. Now, the second thread accesses the same cache-line. It is not in the cache of the thread, therefore it sends a request to the TD. The TD already know that the cache-line is in the cache of the first thread, so it sends a request to that cache. After the request returns the cache-line, the second thread modifies it. After the modification, the first thread accesses the cache-line again. It sees that the cache-line is in modified state. Thus, it sends a request to the TD. The TD forwards the request to the second thread and the thread returns the cache-line. The application repeats this process many times.

If both threads are pinned to some cores, the time of sending data between the cores is the same, even though the TD changes. The only thing that changes with a different TD is the latency of the reading information from the TD.

The Implementation Using False Sharing

The first implementation creates a false sharing between the threads. The false sharing happens, when there are seemingly no related data stored on the same cache-line, and the data are accessed from multiple threads. As the cache-line is the smallest unit in the memory, even if every thread changes only its own private piece of the cache-line, the whole cache-line must be invalidated.

Our program creates a structure with two counters. Each of them belong to one of the threads. The structure is mapped to the cache-line and the threads start incrementing their counters. The implementation does not synchronize the threads. The first thread (a slave) increases the counter in the cycle. The second thread (a master) increases the counter and also measures the code.

This implementation shows worse results than the other one. We assume that the main problem is that the threads are not forced to run simultaneously. It might happen that a thread accesses the cache-line without the interference of the second thread.

Our second implementation uses a *volatile* counter and an *atomic_t* flag. The *volatile* keyword signals to a compiler that it should not optimize the variable. The second variable is the flag to show whose turn is it. It ensures that the threads change.

Both variables are stored on the tested cache-line. A waiting for the turn is performed by a classic busy waiting. One thing to realize is that the cache-line is loaded twice, for the counter and for the flag.

We allocate 1024 pages and then we run the application with pinned threads. We use kthread_bind() to pin a thread to a core. For creating a thread, we use the function kthread_create(). We use the function wake_up_process() for starting the thread. There are two functions - kthread_stop() and kthread_shoud_stop() – for ending the thread. The first function sends a signal to the thread that it should end. The second function checks whether the signal has already arrived.

Sometimes, even though we call kthread_stop() after wake_up_process(), the thread does not start. For this reason, we create our own signals using an *atomic_t* variable that signals the thread. We do not know, why it happens. Either it is due to some compiler optimizations, or it is an internal behavior of the functions.

Threads Numbering

We test multiple cores and multiple cache-lines. First, we pin the core, and then we test all the latencies of the cache-lines in allocated pages.

A side effect of our testing is that it finds logical units placed on the same core. If the threads run on the same core, the state of the cache-line is not synchronized through the TDs, but it is synchronized within the core. The latency in that case is really small.

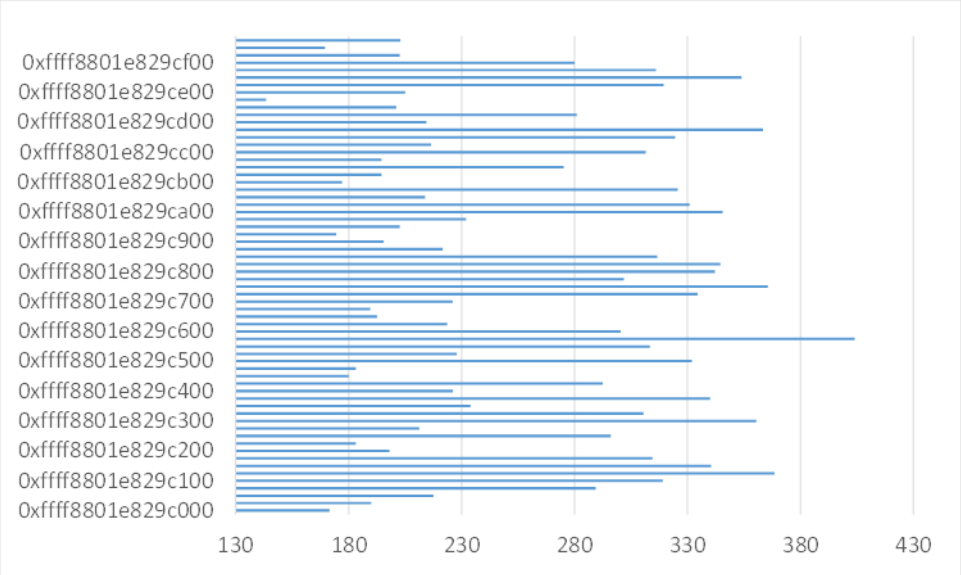
Every physical core has 4 logical cores (threads). We found that the IDs of the logical threads are shifted. The ID = 0 does not belong to the core 0, but to the core 60 (the last one). The IDs of each core are tuples of four. They start with 1 and go to 243. The last number of thread is 0. E.g. (1, 2, 3, 4) are threads of the 1st core, (5, 6, 7, 8) are threads of the 2nd core and (240, 242, 243, 0) are threads of the last 61st core.

The Latencies of TDs

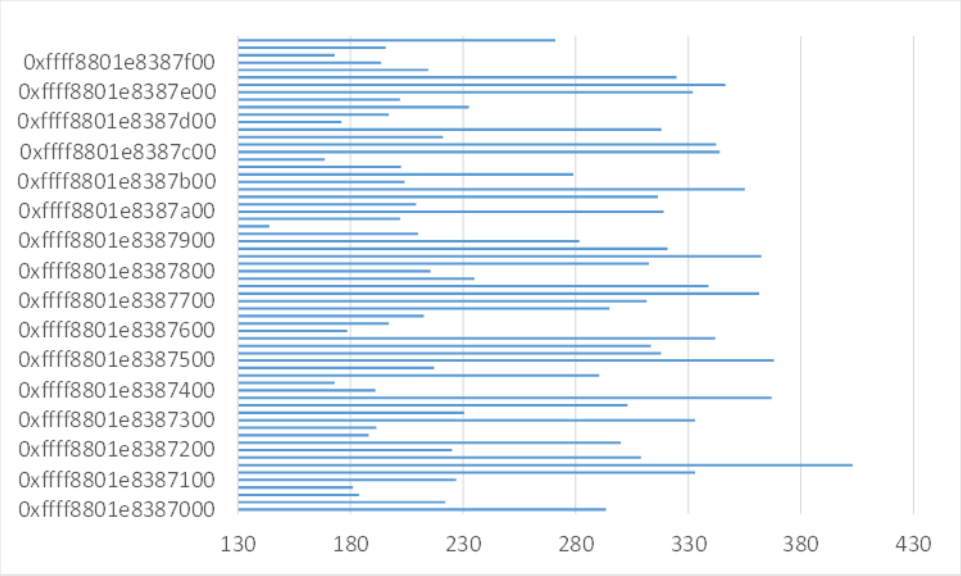
The examples of the results are shown in Graph 10, Graph 11, Graph 12 and Graph 13. Each graph holds the values of one allocated page. Every value shows the latency of the cache-line. We see that the same values of the latencies appear in each graph. The smallest value is around 144 cycles, the highest value is around 402 cycles. The values repeat in 4KB memory cycles, which correlates with the size of pages. Therefore, we assume that each cache-line in every page is mapped to a unique TD.

Unfortunately, we did not find a function for the mapping. There is no obvious pattern, the values look randomly shuffled.

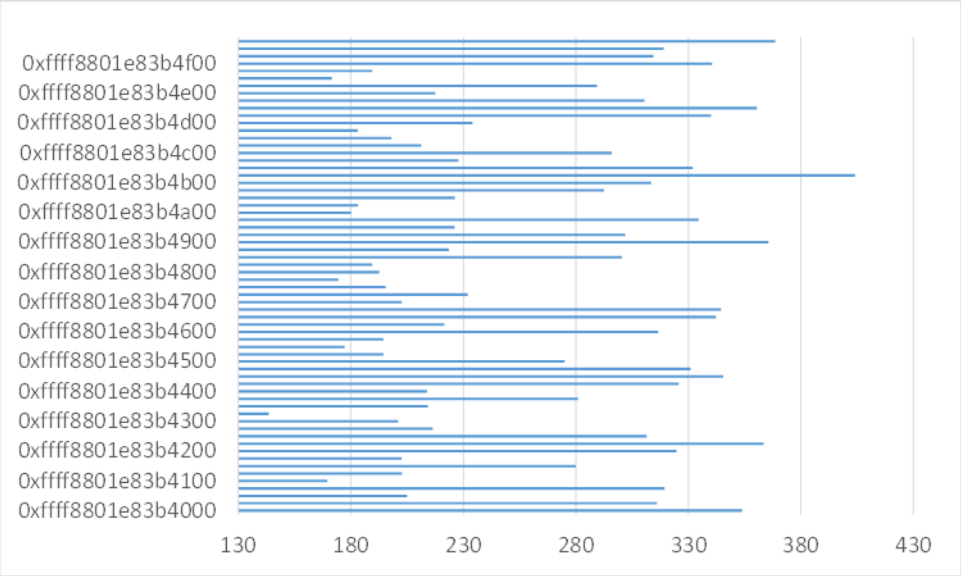
The measured values are normalized. We measure the latency of 128 ping-pongs, which correspond to 512 memory loads. As previously stated, each thread loads 2 cache-lines per a ping-pong. Also, the cache-line is loaded twice, because it contains two stored values. Therefore, all measured values are divided by 512 to get the latencies of a single memory load.



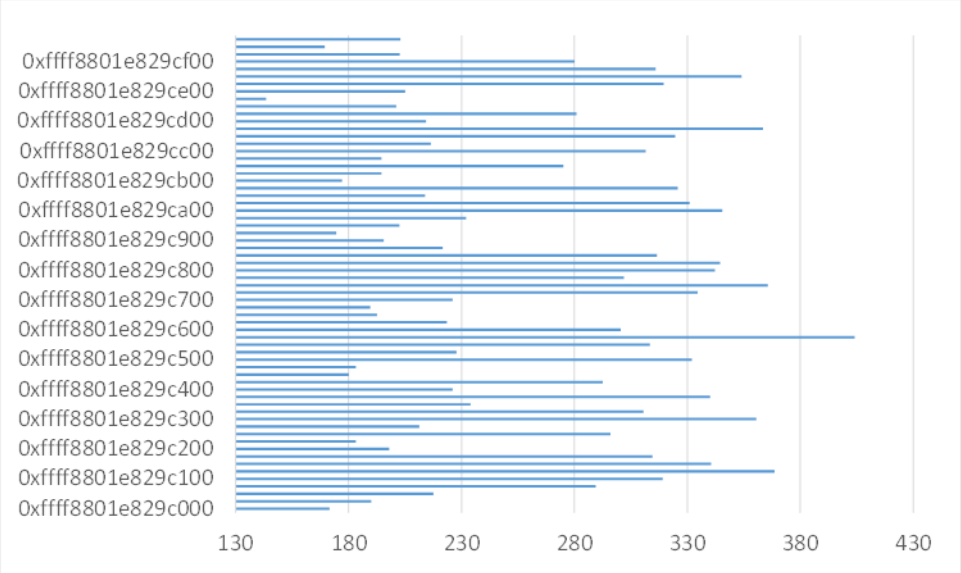
Graph 10: Measuring a latency of a TD



Graph 11: Measuring a latency of a TD



Graph 12: Measuring a latency of a TD



Graph 13: Measuring a latency a TD

NUCA effect

The fastest latency takes around 144 cycles. The slowest latency takes around 402 cycles. This is a huge difference. The ratio of the difference is almost 1:3. The only thing that changes between different cache-lines is the position of the TD. It implies that, if multiple cores share a cache-line, it depends where the cache-line is placed. This effect appears in some multicore architectures. These architectures are called Non-Uniform Cache Architectures (NUCA) [16].

”In an attempt to improve upon conventional “deep” cache hierarchies, a number of Non-Uniform Cache Architectures (NUCAs) have been proposed. In general, a NUCA flattens the conventional multi-level cache hierarchy by using a fewer numbers of cache hierarchy levels with a large number of banks of the same

memory technology (e.g., SRAM, EDRAM, etc.) in each level of the cache hierarchy. As a consequence of the physical structure of such cache architectures, entries in different banks of the same cache memory have non-uniform access times dependent on physical position, giving rise to the term NUCA." [16]

The non-uniform access times are caused by the distributed tag directories (DTDs). If the cache-line is shared between multiple cores, the DTD needs to be used and the performance goes down. Therefore, the Xeon Phi is not exactly NUCA, but it has similar behavior when some data are shared.

If some data are shared between the multiple cores, we should be careful where the data are stored. Sharing almost always implies the usage of the TDs. Because we do not know the mapping between the cache-lines and the TDs, the best way is to benchmark it. We only need to benchmark 64 consecutive cache-lines.

On the other hand, the difference of the latencies shows only for some special cases. The data must fit into a single cache-line and must be shared between two cores. If the data use multiple cache-lines, the cache-lines are uniformly distributed in the TDs around the ring. In that case the latency gets closer to the average latency of the TDs, which is around 260 cycles. If multiple cores access a single cache-line, thanks to the uniform distribution of the TDs, the overall latency of the access of multiple cores goes to the average latency. In the last case, when multiple cores shared multiple cache-lines, the latencies are also close to the average values.

4. Application

This whole thesis is created because of the performance problems of the streaming system Bobox on the Xeon Phi. In this chapter, we describe relevant aspects of the Bobox that are necessary for understanding of the proposed improvements. More detailed information about Bobox can be found in [1], [2] or [7]. We also introduce the improvements to the Bobox. We show that there is no problem with the scheduling and therefore we implement one of the possible improvements to prove it.

4.1. Bobox Overview

There are two basic types of programming languages. The general purpose programming languages (C, C++, Java, Javascript, Python, etc.) and the domain specific languages (UNIX shell scripts, MATLAB, VHDL, SQL, YACC, etc.). A *domain specific language (DSL)* is a computer language specialized to a particular application domain. On the other hand, the *general purpose languages (GPL)* can be used across multiple domains. The *streaming languages* are a subgroup of the DSL.

The idea of the streaming languages is that data can be represented as a stream of tuples. The streams can be processed by *operators/kernels/filters*. The operators transform input streams into output streams. The execution of the language is managed by a runtime. The runtime for the streaming languages is called a *streaming system.* The implementation of the operators and the operator's mutual connections together are called an *execution plan*. An example of the execution plan is depicted in Figure 18.

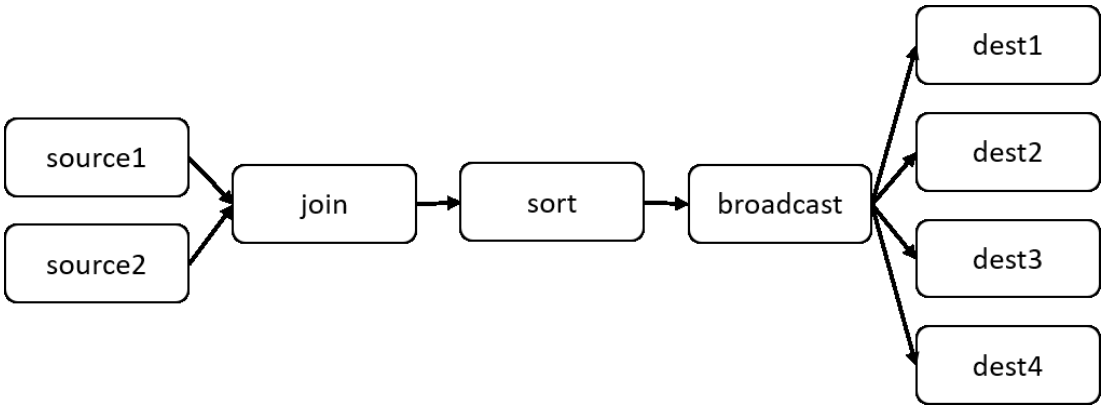


Figure 18: An example of the execution plan

Bobox is a parallelization framework developed at the Department of Software Engineering of the Charles University. It can be considered a streaming system,

because it provides operators known as *boxes*, and their mutual connections, known as an *execution plan* or a *request*. The boxes process data streams. They transform the input streams into the output streams. The stream is a flow of envelopes. The *envelope* is a sequence of some tuples. The envelopes travel between the boxes. When an envelope arrives into the box, the box is scheduled by a task scheduler. Later, the scheduled box is executed by a thread. The whole execution plan is evaluated this way. Except for the data envelopes, there is a special management envelope. It is called the poisoned pill. The poisoned pill is sent after the last data envelope and it denotes the end of the stream.

One of the advantages of the streaming systems is that they naturally introduce *inter-operator parallelism*. Each of the operators can be executed in parallel in case that they do not depend on each other. It is up to the runtime, how the operators are executed. Besides the inter-operator parallelism, the Bobox has introduced *intra-operator parallelism*, i.e., the single operator is adjusted to be able to run in parallel.

4.1.1. Task Scheduler

The task scheduler is responsible for the execution of the boxes. The box is scheduled and executed by a thread when the input envelopes are ready for processing. The Bobox cannot analyze the implementation of the boxes, hence it knows nothing about the fact how the data are accessed inside the box. It presumes the input envelopes to be hot in cache, therefore it prefers to run the current box on the same processing unit as the previous one.

From the view of the scheduler, each box is in one of three states:

- **waiting** – The box is waiting for the input envelopes.
- **scheduled** – The input envelopes are prepared and the box is ready to be executed.
- **running** – The box is being executed.

When the state of the box switches to scheduled state, a new task that executes the box, is created and passed to the scheduler.

There are two types of tasks:

- **immediate** – The task is associated with the processing of the input data. It is expected to be executed as soon as possible, preferably by the thread which created it.

- **deferred** – The task is directly associated with no data. It can be executed by any thread.

Both types of tasks should be executed as soon as possible and also as close as possible to the thread that spawned the task. It is obvious that each task belongs to exactly one execution plan. Therefore, the Bobox runs the tasks from different execution plans on a different physical processor.

Task Scheduling Algorithm

At the beginning of the Bobox, the host system is analyzed. The configuration of the system – e.g. NUMA factor, the hierarchy of the cache – is set according to the analysis. The processing units that share at least one level of a cache form a processing group. Each group contains one thread pool. Each pool has the same amount of threads as is the number of processing units inside the group. The affinity of the threads is set to this group. Each processing unit has one queue of immediate tasks. All processing units inside the processing group share one queue of deferred tasks.
The distance of the units within a group is the lowest level of the cache shared between the units. The distance of two groups is the distance of their corresponding NUMA nodes, i.e., it is equal to the NUMA factor.

The immediate task queue is an ordinary queue. The deferred task queue is a more complex data structure. It holds the tasks of different execution plans separately. Also, the deferred tasks are sorted from the youngest task to the oldest one.

When a thread is searching for the next task, the scheduler chooses the task according the first applicable rule:

1. The youngest task from the queue of immediate tasks of the processing unit of the thread is taken.
2. Other processing units of the same group are scanned (in increasing distance) and the first non-empty immediate queue is found. If such queue exists, the oldest task is taken.
3. The youngest deferred task of the oldest requests from the shared queue of deferred tasks of the current group is taken.
4. Other processing groups are scanned (in an increasing distance) and the first non-empty shared queue of deferred tasks is found. If such queue exists, the oldest deferred task of the second oldest request is taken. If the queue has tasks of only one request, its oldest deferred task is taken instead.

5. Immediate queues of the processing units from other groups are scanned. If non-empty queue is found, the oldest task is taken. The immediate queues are scanned in round-robin manner and the thread remembers the last non-empty queue found. When this rule is applied again, the scan resumes where it previously ended, i.e., when the thread steals a task from another processing unit, it tries to steal a task from the same unit again. It may have the context of the hot in cache.
6. If all steps above fail, i.e., there is no available task to execute, the thread is suspended.

4.1.2. Memory Allocator

Default memory allocators in C/C++ do not work well with the envelopes of the Bobox, because they do not scale well in parallel environment. Therefore, the Bobox has introduced its own NUMA-aware allocator for envelopes in order to achieve a better performance.

Blocks of memory larger than 512KB are accessed rarely, because the preferred size of an envelope is smaller than 256KB. For this reason, these blocks of memory are allocated and deallocated using standard system functions for allocating/deallocating virtual memory (`mmap`/`munmap`, `VirtualAlloc`/`VirtualFree`).

For the blocks of memory between 8KB and 512KB, there are several fixed-size block sub-allocators. When a new block is requested, the allocator chooses the appropriated sub-allocator – the one with the smallest size of blocks larger or equal to the requested size. This sub-allocator returns a free block. If there is no such block, it creates a new super block that is used as an additional memory pool. The free blocks are kept in a linked list. This linked list is served in LIFO manner.

Each allocated block has a small header with the information about its sub-allocator and its super block. The super blocks are deallocated when all underlying memory blocks are deallocated. The allocation and deallocation of super blocks is managed by the super block allocator. Each super block allocator belongs to one NUMA node that uses it for memory allocations.

Blocks of memory smaller than 8KB are allocated similar way as the bigger blocks. Moreover, there are some additional optimizations to reduce the internal fragmentation and the overhead of the management of small blocks.

4.1.3. Intra-operator Parallelization

The intra-operator parallelization is difficult to achieve automatically in streaming systems because the operators are designed as black boxes. One of the possibilities, which are also used in the Bobox, is the duplication of the operators. This approach corresponds to the data parallelism [17].

The Bobox distinguishes between two types of operators – *stateless operators* and *stateful operators*.

Stateless Operators

The stateless operators do not maintain their inner state. The results are independent of the previous results. The example of stateless operator is a filter operator. It filters envelopes based on a condition that is evaluated for each envelope independently.

The stateless operator is easy to parallelize. It splits the input streams into multiple sub-streams that are processed in parallel. The results of processing are later joined together into a single output. Figure 19 shows the example of the parallelization of the stateless operator. The dispatch operator splits the input stream into multiple sub-streams and it sends them into the output streams according to a rule, e.g. round-robin rule. The consolidate operator is the inverse operator to the dispatch operator. It forwards envelopes according to a rule, e.g. round-robin, from the input streams into the single output stream.

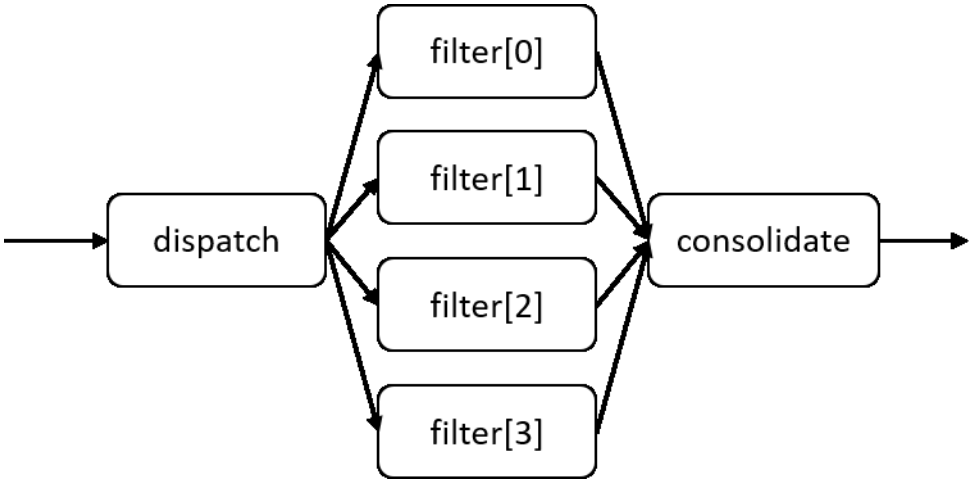


Figure 19: The parallelization schema of the stateless operator

Stateful Operators

The stateful operators need to maintain their inner state. The result of the operator can depend on every previously processed tuple. An example of the stateful

operator is the merge join operator. The merge join operator performs a join algorithm on relations sorted by the join attribute. It has two inputs – the left one and the right one. It reads the input streams and finds the sequences of the same values of the join attribute in the left and in the right input. Then it creates a cross product of those sequences. The merge join and other algorithms from this chapter are described preciously in [2].

One of the problems of the stateful operators is that not all of them can be easily parallelizable. The Bobox supports a parallelization of the group of operators whose algorithm match the schema from Algorithm 2. Bobox calls these operators parallelizable. One important aspect of the parallelizable operators is that the update of the state is separated from the processing of the envelopes.

```
state = initial_state();
while(not_end_of_stream()) {
        envelope = get_next_envelope();
        process_evelope(envelope, state);
        state = update_state(envelope, state);
}
```

Algorithm 2: The schema of parallelizable operator

The concurrency is achieved by the replication of the operator. The schema of the parallelization of the parallelizable operator is depicted in Figure 20. The schema is similar to the schema of the stateless operator. The only difference is the usage of the broadcast operator instead of the dispatch operator. The broadcast operator forwards all the envelopes from the input into all its outputs.

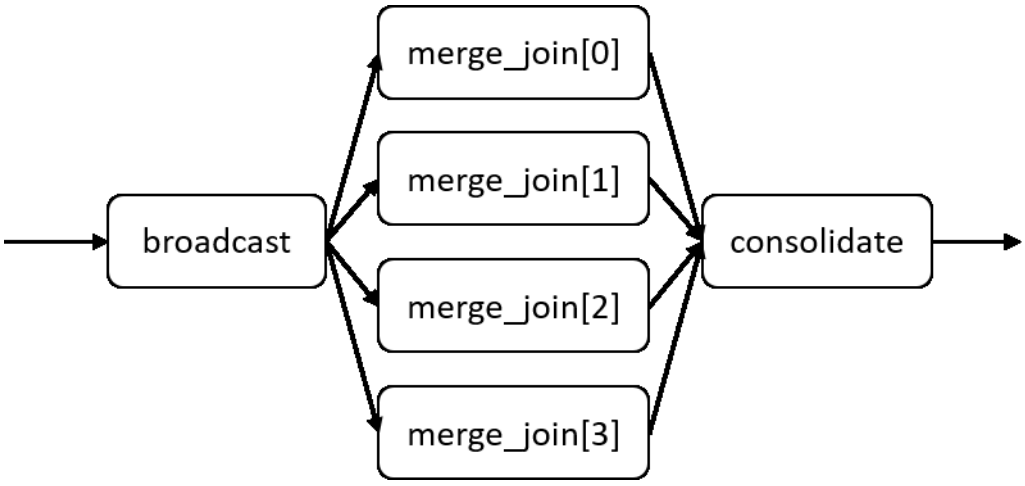


Figure 20: The parallelization schema of the parallelizable operator

Each replicated operator keeps its own copy of the state. Every time it receives an envelope, it updates the state. Also, it performs its part of the work when an appropriate envelope arrives. The algorithm performed by each replica is shown in

Algorithm 3. The RID stands for Replica ID; i.e., a unique identifier of each replica. The number of all replicas is N.

```
state = initial_state();
phase = 0;
while(not_end_of_stream()) {
        envelope = get_next_envelope();
        if (phase % N == RID)
                process_envelope(envelope, state);
        state = update_state(envelope, state);
        phase = phase + 1;
}
```

Algorithm 3: The algorithm performed by each replica.

There are also other improvements to the Algorithm 3. The first improvement is used, if it is faster to update the state at once for N-1 envelopes than to perform N-1 updates separately. The adjusted algorithm is described in Algorithm 4. For example, this schema is used in parallel multi-way merge operator in Bobox.

The idea behind the second improvement is following: An envelope is processed only by one replica, while the remaining N-1 replicas use the envelope to update their inner states. That implies that the remaining replicas are performing exactly the same operations. The redundancy is eliminated by dedicating a preprocess operator. It calculates the values of a state for each envelope and it includes this information inside the envelope. The algorithm of a preprocess operator is described in Algorithm 4. The corresponding algorithm for replicas is in Algorithm 5. This schema is used for parallelization of the merge join operator.

```
state = initial_state();
while(not_end_of_stream()) {
        envelope = get_next_envelope();
        store_state_to_envelope(envelope, state);
        send_envelope(envelope);
        state = update_state(envelope, state);
}
```

Algorithm 4: The algorithm of the preprocess operator

```
envelope_index = RID;
while(not_end_of_stream()){
        envelope = get_nth_envelope(envelope_index);
        state = envelope.get_state();
        process_envelope(envelope, state);
        envelope_index = envelope_index + N;
}
```

Algorithm 5: The algorithm performed by each replica when prepended by the preprocess operator

There are multiple models of parallelization of boxes in the Bobox. The model says how much the operator can scale. Naturally, there is a serial model. It does no intra-operator parallelization. Another model is a parallel model. It introduces two options for the inter-operator parallelization. Both options adjust the number of duplicated operator inside the inter-operator parallelization. The first option (denoted as * in Bobox) duplicates the operator to the number of threads per a NUMA node. The second option (denoted as ** in Bobox) duplicates the operator to the number of processing units in the system. For example, the SMP16 solution contains two NUMA nodes and 8 threads per a node. It creates 8 copies of the box with the first option (*) and 16 copies of box with second option (**).

4.1.4. SPARQL Engine

As mentioned previously, the Bobox is only a framework. One of the applications that were created with the aid of the Bobox is in-memory SPARQL engine for querying RDF data. The engine is used for the evaluation of performance of the Bobox. More information about the SPARQL engine can be found in [18].

The Resource Description Framework (RDF) [19] is a data model for description of information. The data are stored in the form of triples. The triple consists of a subject, a predicate and an object. The subject is the resource and the predicate expresses a relationship between the subject and the object. For example, the notion “This thesis discusses Xeon Phi” can be represented in RDF as the triple: (subject) “this thesis” – (predicate) “discusses” – (object) “Xeon Phi”.

SPARQL is a popular SQL-like query language for RDF data. A SPARQL query can consist of multiple triple patterns, logical symbols and variables. The SPARQL engine searches for the sets of triples that match the triple patterns and binds the variables to appropriate values. An example of a single query which returns all the

topics for all thesis of the given author is written in Algorithm 5. The symbol "*?*" denotes a variable.

```
SELECT ?topic
WHERE {
        ?thesis author ?author .
        ?thesis discuss ?topic .
}
```

Algorithm 5: An example of the SPARQL query

The SPARQL language provides a set of multiple operations known from SQL such as JOIN, SORT, SCAN, UNION, etc. All these operations must be implemented by the engine. While the Bobox is being used for implementation of the SPARQL engine, the operations are implemented inside the boxes (operators) and the triples are stored as the tuples inside envelopes. After a query is received by the engine, it gets compiled and optimized. After a validation of the query, the engine prepares the query execution plan. Then, the execution plan is executed over the loaded RDF data by the Bobox.

4.2. Bobox Improvements

In this part of the chapter, we test the proposed improvements in the Bobox framework. We prove that the main performance problem is not caused by the scheduler. We locate the main problems and propose the solution that solves them. Unfortunately, the implementation of the solution to the problems is beyond the scope of this thesis.

4.2.1. Testing Improvements

We perform multiple experiments with the Bobox. We use the SPARQL engine and the SP^2Bench benchmark [20] with its testing 5M and 1M datasets. Additional to the datasets, the benchmark contains several SPARQL queries. The queries are transformed to appropriate execution plans by the SPARQL engine. This way, we can easily test the performance of the Bobox. All used queries with their execution plans can be seen in Graph 14. We use the same queries as in [2] in order to know reference results.

We use two types of Xeon Phi for the experiments. We test the Xeon Phi with architecture Knights Corner (denoted as KNC) and the newer version of Xeon Phi with architecture Knights Landing (denoted as KNL). By using those two cards with

different architecture, we can distinguish the problems of the architecture from the general problems of the Bobox. Also, the KNL allows the run standard 64bits applications which make the debugging and profiling of the programs easier compared to the same work on the KNC. Therefore, if the architecture is relevant for the results, we state it explicitly. If the architecture is irrelevant, we use the results of the KNL.

As a reference solution, we use a blade server containing 2x4-core CPU (with Hyper-Threading) and 96GB RAM in SMP configuration. We denote it as SMP16. The results of the reference solution are shown in Graph 14. The results correspond with the results from [2] as is expected.

The configuration of the Bobox is set based on the Bobox default analyzation. All tests use the default parallel model of operators with *; i.e., the number of duplicated operators corresponds to the number of threads per a NUMA node. The default setting for SMP16 is two NUMA nodes and each with 8 threads. The default setting for the KNL is 256 threads with 1 NUMA node. For the KNC, the default setting is 240 threads with 1 NUMA node.

4.2.2. The Preparation of the Execution Plan

We started our investigation with the repetition of the experiments for which we already had results measured on SMP16. As we can see in Graph 14, the results on KNL are much worse than on SMP16. The time is measured by the Bobox. The Bobox is able to measure the time of the evaluation of the query. This time does not include the time of a SPARQL runtime initialization and neither the compilation of the query.

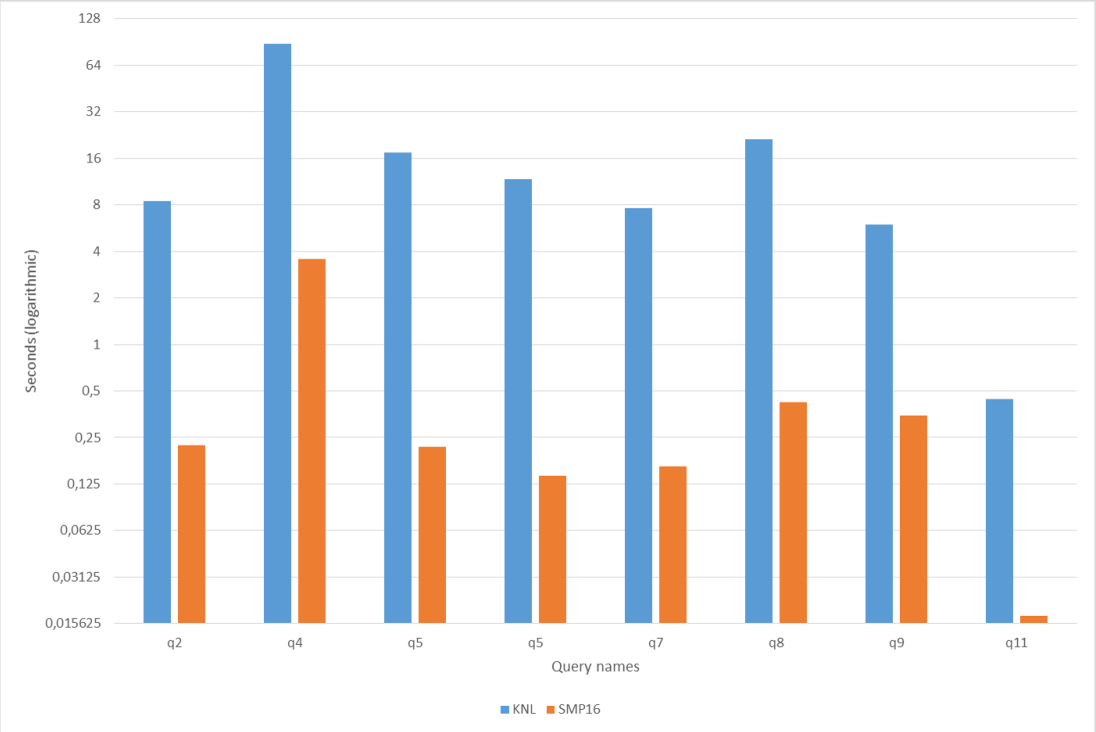
Closer look shows that the evaluation runs in two phases. The first phase prepares the execution plan. It creates all the boxes and the connections between them. It has to link the consecutive boxes together and create input and output buffers. The second phase is a real evaluation of the plan. Therefore, we create our own measurement to make the results of the experiments more precise. We measure each phase independently.

As we can see in Graph 15, the biggest part of the time of the evaluation takes the preparation of the execution plan. Other experiments depicted in Graph 16 and in Graph 17 show that the time of the preparation grows linearly with the number of requests and also with the number of threads.

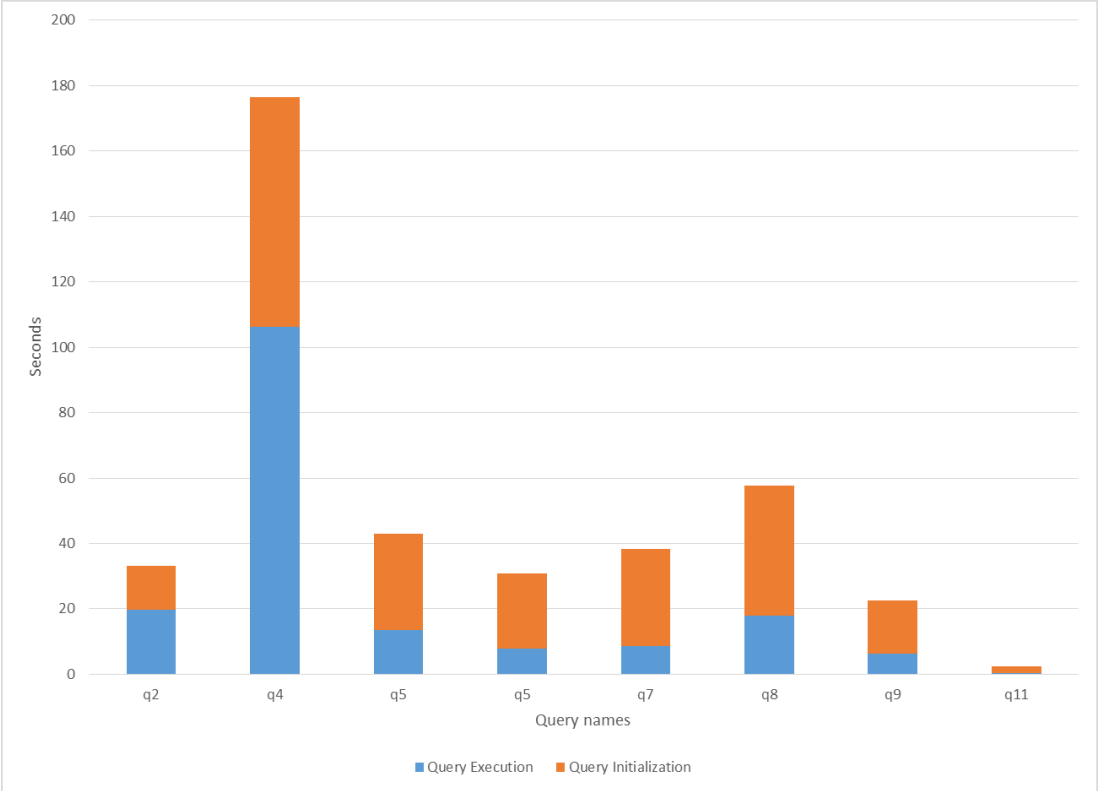
The linear growth with the number of requests is predictable. The Bobox needs to create a new execution plan for each query. The linear growth with the number of threads is not so obvious at first. However, the reason for that is the intra-operator

parallelization. Following our earlier example from 4.1.3, we know that the parallelization duplicates the operators. In case of the parallel version (*) of the operators, it means that there is one operator per a thread. The results with the serial version of the operators in Graph 18 prove our assumption. It implies that there is an initialization for each duplicated operator. For a large number of duplicated operators (the number correspond to the number of threads per a NUMA node), the time of initialization of the operators starts to be significant.

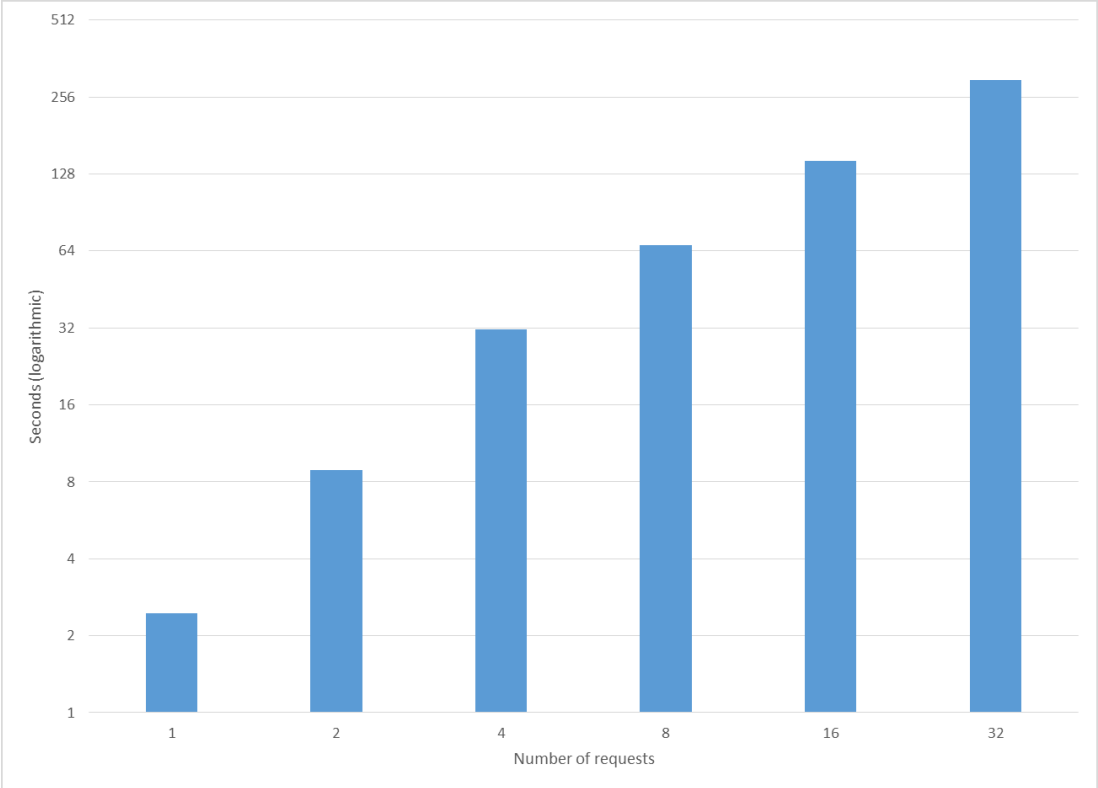
At this moment, the initialization of the execution plan is single-threaded. That implies one of the possible improvements – make the Bobox multi-threaded. More sophisticated improvements would probably require the change of whole architecture of the Bobox. However, both improvements are beyond the scope of this thesis.



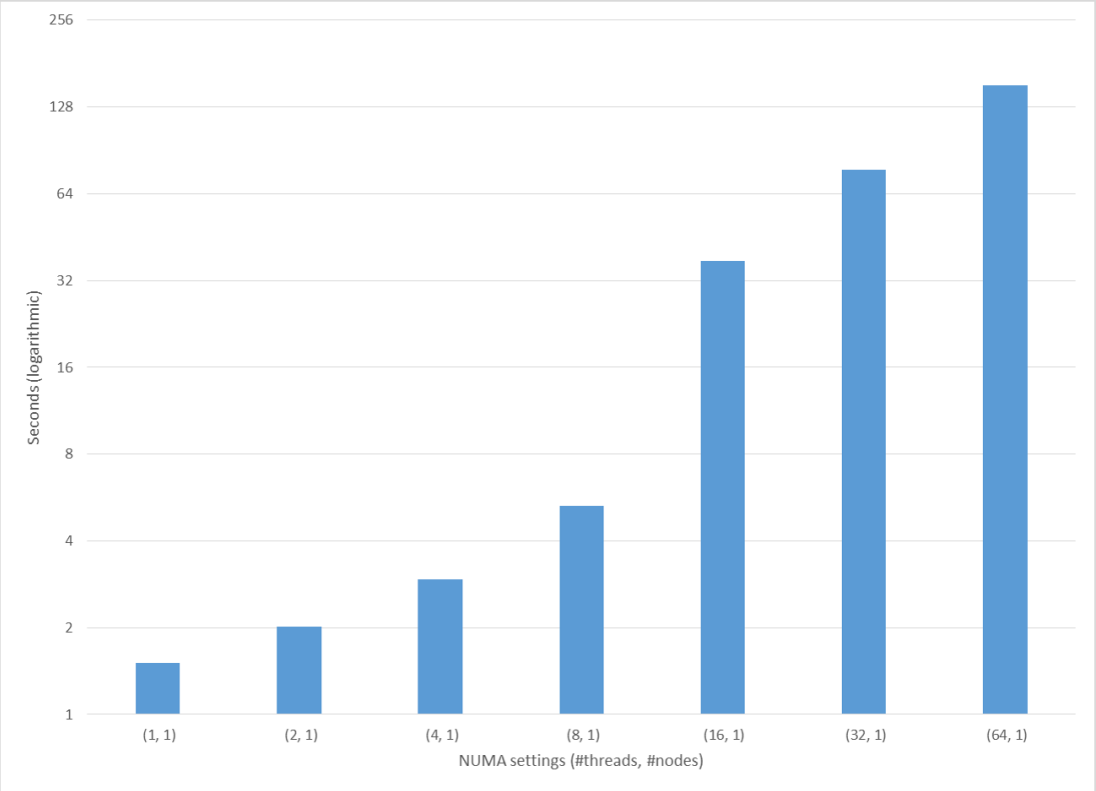
Graph 14: The evaluation time of single queries on the KNL and on the SMP16 (1 request)



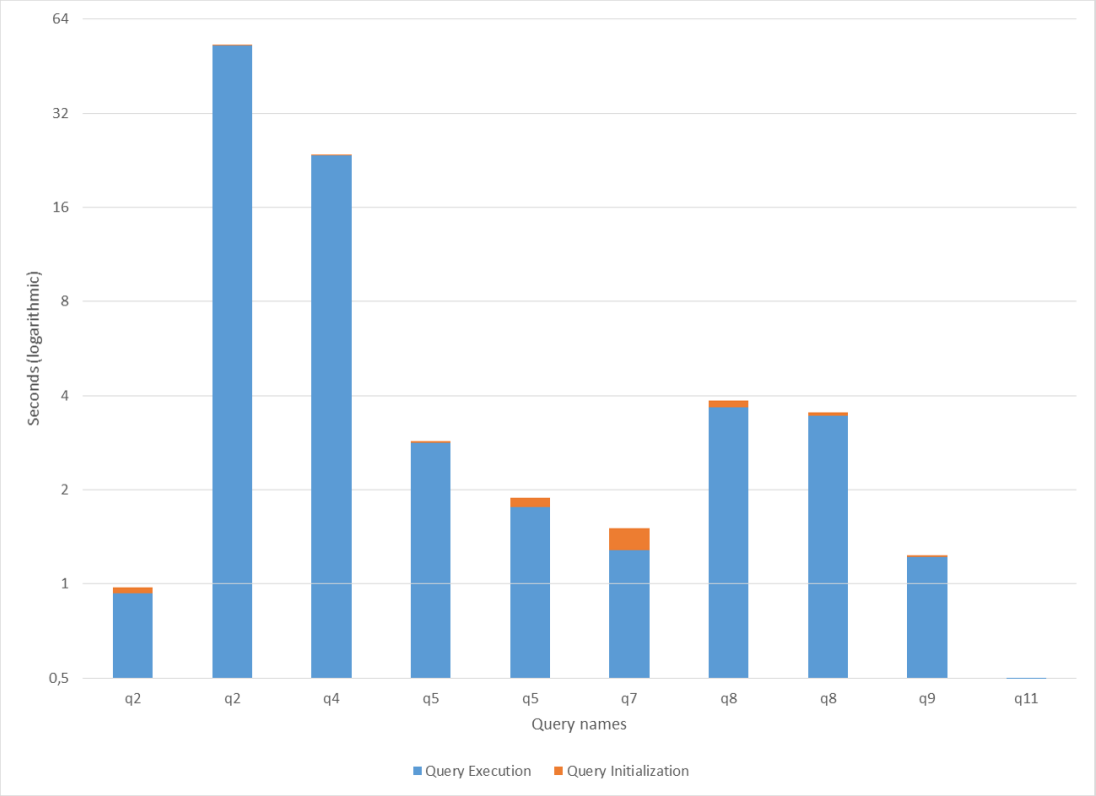
Graph 15: The evaluation time of single queries with parallel version of operators on the KNL (4 requests)



Graph 16: The preparation time for different number of requests



Graph 17: The preparation time for different number of threads (32 requests)

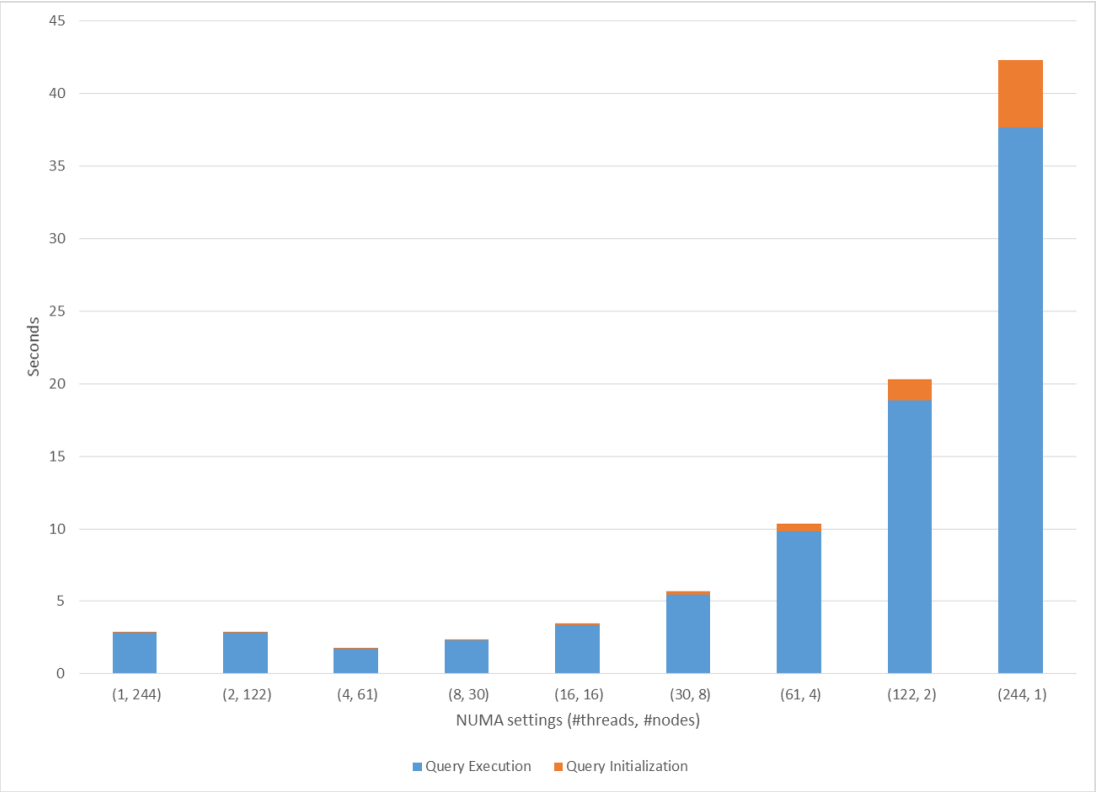


Graph 18: The evaluation time of single queries with serial version of operators on the KNL (4 requests)

4.2.3. Testing NUMA on the KNC

We showed with the benchmark in 3.3 that the performance of application could be tuned with the right settings of NUMA nodes. Therefore, we test the KNC with different settings of NUMA nodes. The results are shown in Graph 19.

As we can see, the results are highly influenced by the bad scaling of the broadcast operator. Therefore, it is hard to say, whether the NUMA setting makes sense. On the other hand, we see that the NUMA effect is not big enough to change the results significantly. It corresponds with the fact that the NUMA factor is really small. Thus, we should take this improvement into consideration only if all major problems are solved.



Graph 19: The evaluation time of query for different NUMA settings on KNC

4.2.4. Broadcast Operator

There are two types of boxes in the Bobox. The first type of the boxes was shown in 4.1.1. Those boxes can be scheduled and they process the data in a complex way. They are called *basic boxes*. The other type of boxes – *simple boxes* – cannot be scheduled. Their main purpose is to create a model of an execution plan. They hold the information about predecessor and successor boxes of a box. Therefore, if the predecessor boxes push an envelope into the simple box, the simple box simply

forwards the envelope into the successor boxes. Simple boxes may contain a processing of the envelopes, but it should not be time-consuming.

However, this approach is suitable for platforms with small number of threads. It creates a bottleneck together with the automatic intra-operator parallelization (4.1.3).

The functions that insert envelopes into the input queue of the merge join box are one of the hotspots shown in the profiler. As was said previously, the parallel merge join box contains preprocessor boxes which are followed by broadcast boxes. Each broadcast box sends envelopes into duplicated merge join boxes. The number of duplicated boxes depends on the chosen model, i.e. serial, parallel (*) or parallel (**). Eventually, the merge join boxes process the data and send the data into a consolidate box.

There is a problem with intra-operator parallelization using a broadcast box. The problem appears with a big number of duplicated operators, i.e. on the platforms with lots of threads. As the broadcast box is not schedulable, it must create multiple copies of the envelope and push them into the next duplicated operators. This process starts to be really time-consuming for a big number of duplicated boxes. One of the problems is that the pushing an envelope into the box is not so fast on its own. In order to push an envelope into the box, you must acquire a lock for the queue at first, and then you are able to store the envelope. The box is loading the data on the other side of the queue; therefore the queue must be thread-safe.

What happens on the Xeon Phi is following: The preprocessor box is processing the stream of the envelopes. If the envelope is processed, it sends the envelope to the next box. If it were the schedulable boxes, it would just insert the envelope into the queue. Nevertheless, the broadcast box is not schedulable, so it creates copies of the envelope and sends it to its all outputs. After all the outputs receive the envelope, i.e. the envelope is successfully stored in all buffers, the function, which sent the envelopes, can successfully return. This whole time is the preprocessor box blocked. Only after the function returns, it can continue its preprocessing.

To prove our theory, we run the query E2 that tests a scalability of the merge join box. This query is taken from [18] and it is shown in Algorithm 6. It contains only one merge join box after a compilation. We measure the time of broadcasting an envelope into all its outputs. The evaluation of the whole query takes 0.310 seconds. The broadcasting time takes 0.261 seconds. It means that 85% of time is spent on broadcasting envelopes and not on processing the data.

```
SELECT ?article1 ?article2
WHERE {
        ?article1 dcterms:issued ?year.
        ?article2 dcterms:issued ?year
        FILTER (?article1 = ?article2)
}
```

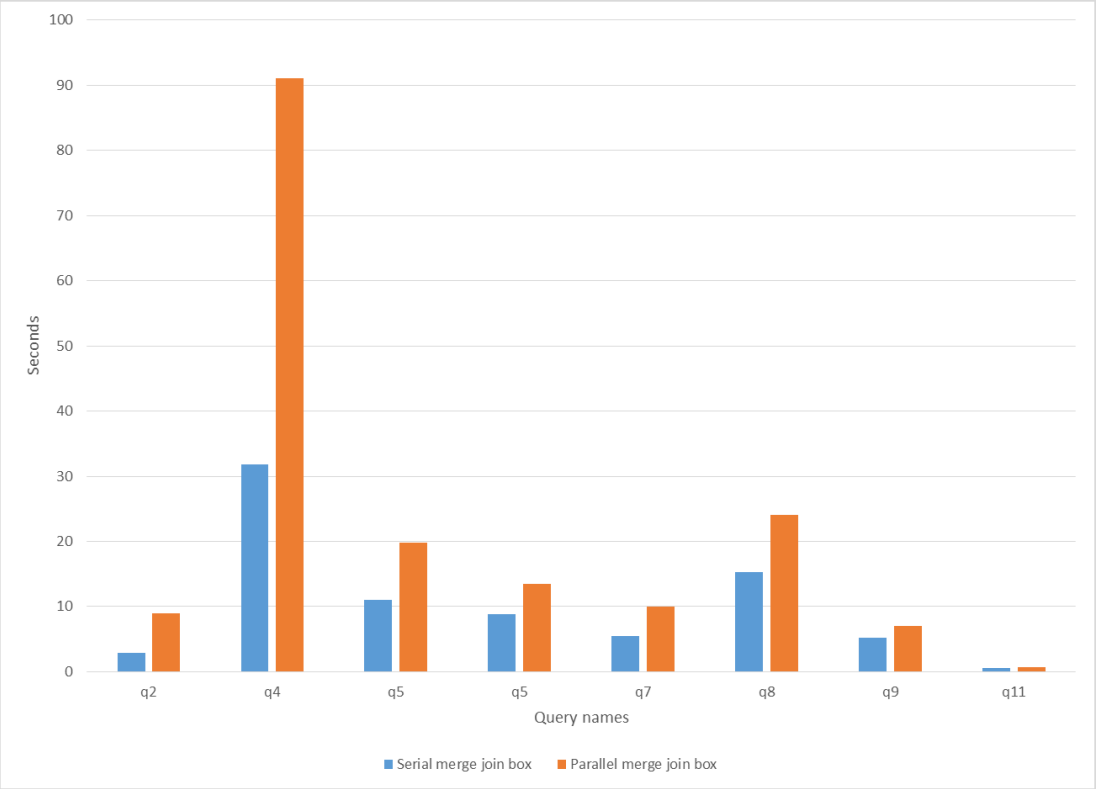
Algorithm 6: The query for testing a scalability of merge join box.

In next test, we replaced the parallel implementation of the merge join box with the serial implementation. The results are depicted in Graph 20. It shows that the results of the serial version of merge box are always better than the results of the parallel version or they equal. The difference depends on the amount of usage of the merge join.

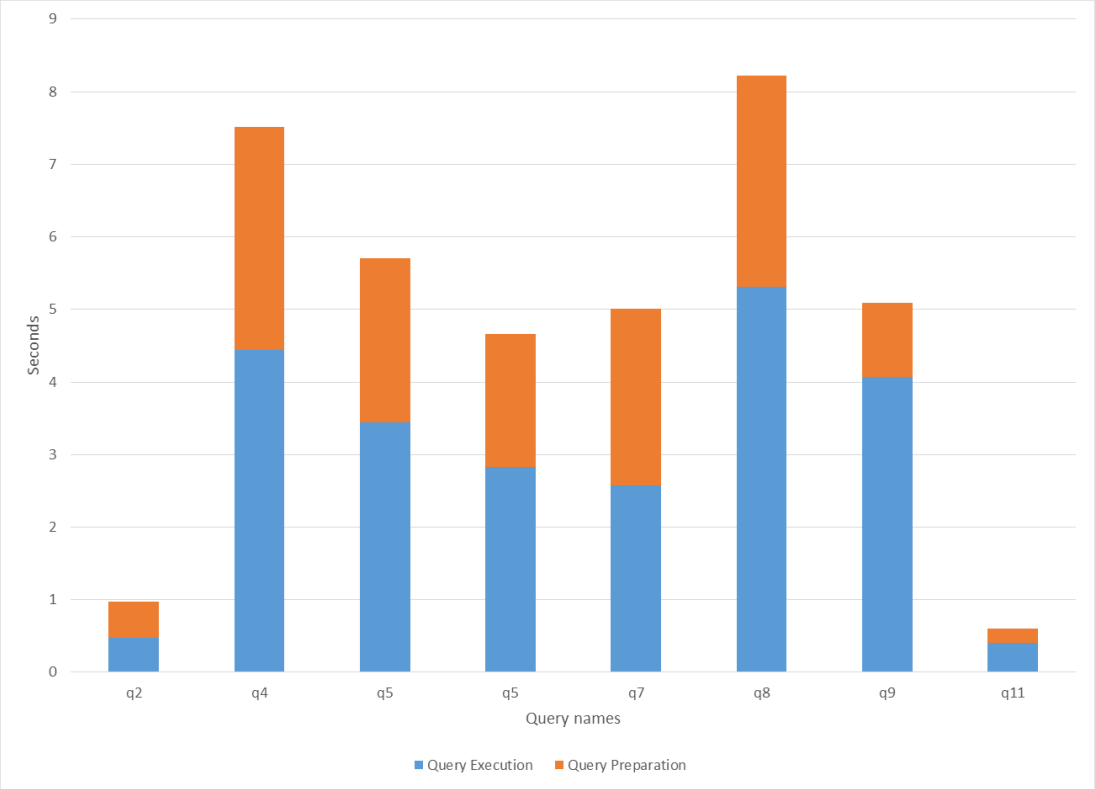
Improvements

We see that the problem of the broadcast box has a big impact. It is caused by the fact that the broadcast box is a foundation stone for the intra-operator parallelism. One small improvement is to create a broadcast box that can be schedulable. We call it *schedulable broadcast box*. The only difference between the schedulable and the original (*unschedulable*) broadcast box is obvious – the schedulable box can be scheduled. Except for that, the implementation of the boxes is the same. This approach should solve the problem with a blocking of the predecessor boxes. The results are shown in Graph 21 and Graph 22. The comparison of the results between the schedulable and unschedulable broadcast on KNL and SMP16 are shown in Graph 23, Graph 24, Graph 25 and Graph 26. We clearly see that the speed of all queries was improved on all testing architectures.

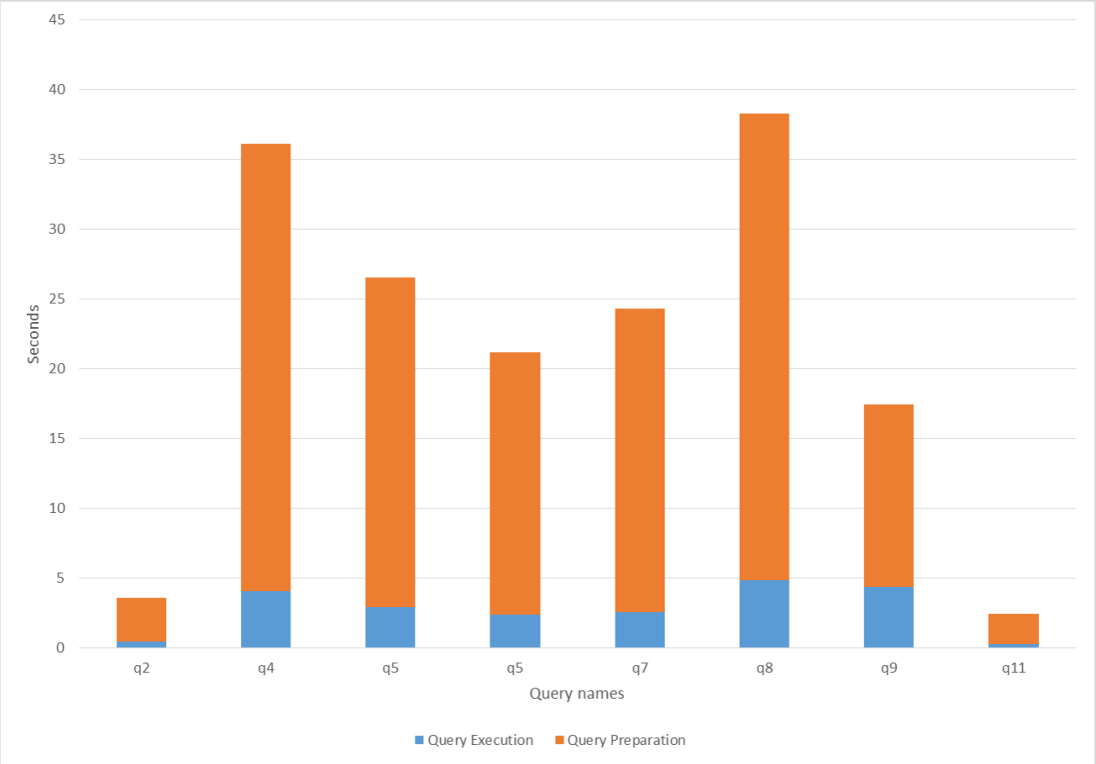
Another idea works on the assumption that the broadcast box scales well for small number of threads ([2]). Thus, we can assign the pushing of the envelopes to multiple threads. However, it breaks the basic premise of the Bobox that the boxes are single-threaded. To overcome this restriction, we need to introduce a new inter-operator parallelization or change the architecture of the Bobox. Unfortunately, that is beyond the scope of this thesis.



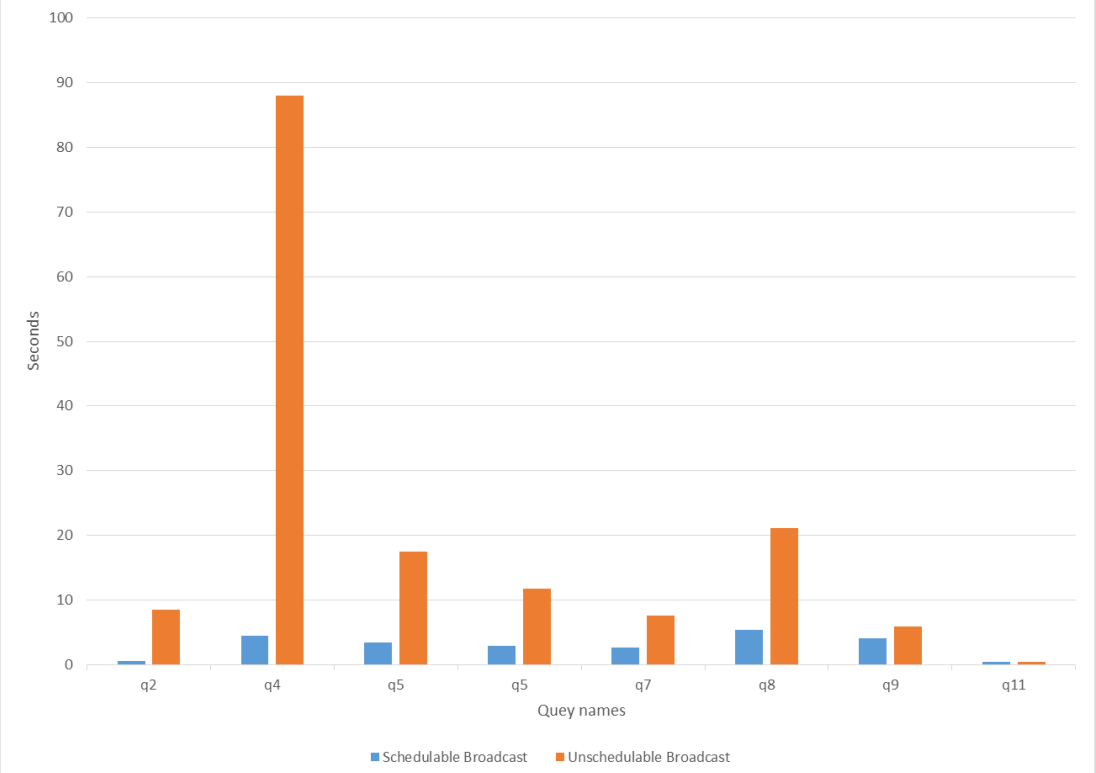
Graph 20: The test of queries with serial merge join box



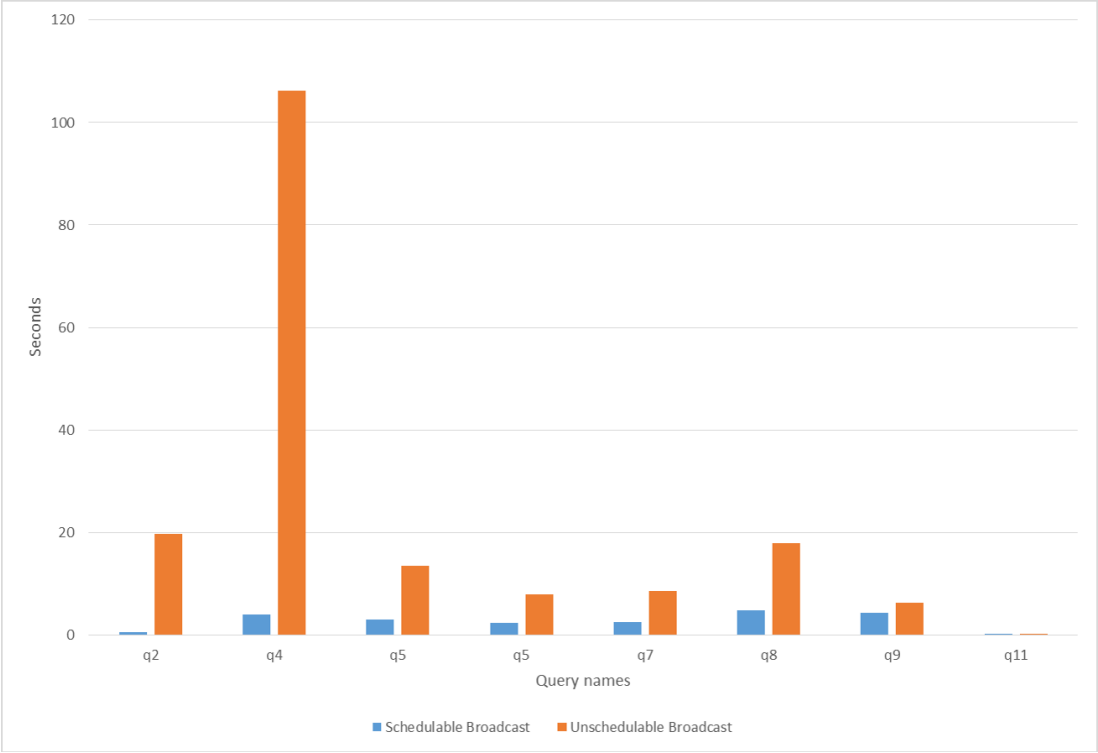
Graph 21: The evaluation time of single queries with schedulable broadcast operators on the KNL (1 request)



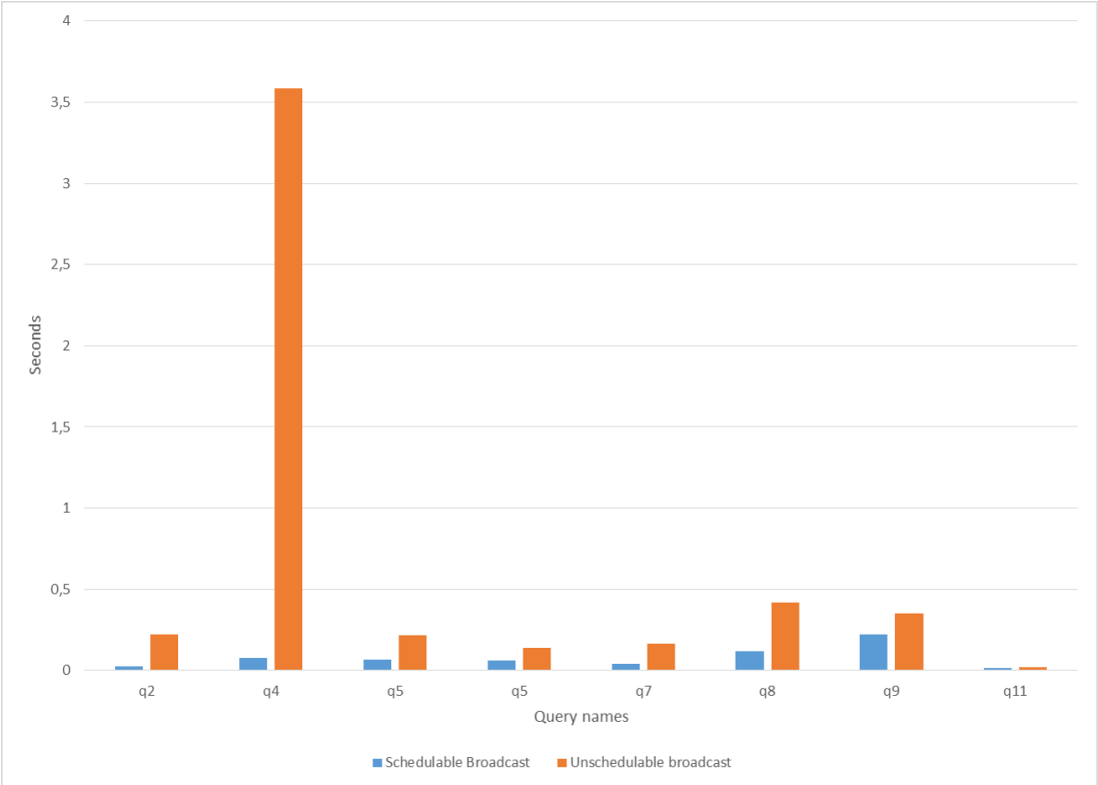
Graph 22: The evaluation time of single queries with schedulable broadcast operators on the KNL (4 requests)



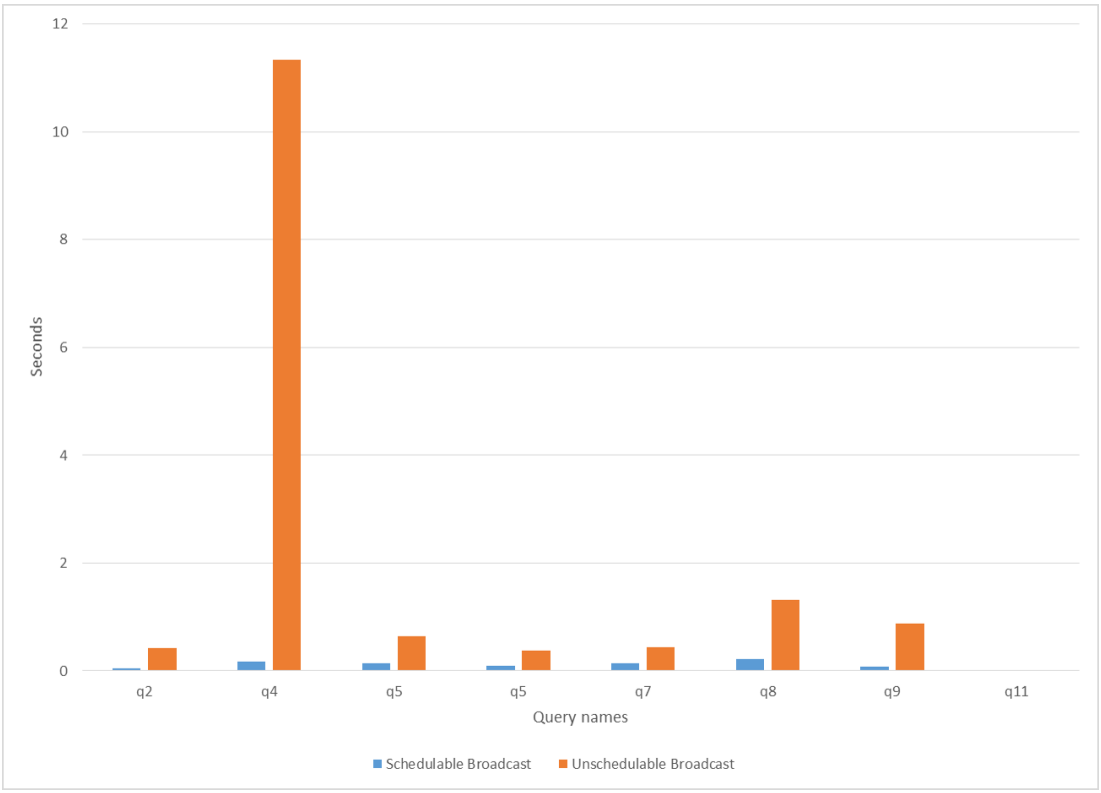
Graph 23: Comparison of single queries between schedulable and unschedulable broadcast operators on KNL (1 request)



Graph 24: Comparison of single queries between schedulable and unschedulable broadcast operators on KNL (4 requests)



Graph 25: Comparison of single queries between schedulable and unschedulable broadcast operators on SMP16 (1 request)



Graph 26: Comparison of single queries between schedulable and unschedulable broadcast operators on SMP16 (4 requests)

5. Conclusion

At first, we made a summary of all available information about the internal architecture of the Xeon Phi. We focused on core and memory, because these pieces of information are relevant to every scheduler. Based on the found results, we proposed additional tests to complete the missing information.

We showed a detail about the positions of the cores on the connection ring. We also confirmed a proposed mapping between physical addresses and memory controllers. These things are important to show a NUMA effect. Given the fact that the cores are placed around the connection ring, there is a difference between accesses to different memory addresses. With proposed four NUMA nodes, there is a small NUMA factor around 1.1. By taking this fact into the consideration, we can gain additional performance boost for HPC application. However, this effect is insignificant to majority of applications.

We found one important memory effect that should be taken into consideration. There is a big difference in accessing the shared memory between different cores. The information of shared memory is kept in DTD (TDs). While accessing a cache-line, the information needs to be read from appropriate TD. Those TDs are also distributed around the connection ring. It implies that there is a difference in accessing into a cache-line from different cores. The difference in latency is around 1:3 ratio. Even though we were not able to find a mapping between TDs and cache-lines, we showed at least that consecutive cache-lines in 4KB pages are assigned to different TDs. A big performance boost can be achieved by incorporating this information into applications. This problem appears only if a single cache-line is shared between two cores. If multiple cache-lines or multiple cores are used, the effect disappears thanks to a uniform distribution of TDs around the connection ring.

The proposed NUMA configuration of the Xeon Phi (KNC) was tested using the Bobox SPARQL engine. Unfortunately, there are other problems that are not connected to the problem with a scheduling. To prove this speculation, we found the real problems. A creation of the execution plan from a query is the biggest bottleneck, because it is single-threaded. However, this problem is more closely connected with the SPARQL engine. We also found that the intra-operator parallelization introduced by [18] does not scale well, because of the bad design of the broadcast operator and other operators. Finally, we introduced some possible improvements and we tested

their benefits. The results show that the performance gets better on all tested platforms – the Xeon Phi and SMP16.

5.1. Future Work

A future work should start with solving problems with the operators in the Bobox. There is a possibility to introduce a new intra-operator parallelization or to change the architecture of the Bobox. We should not forget about the slow initialization of Bobox, because the users are mostly interested in the performance of the whole application.

There are still missing some details about the architecture of the Xeon Phi (KNC), e.g. the mapping between cache-lines and DTD. Another interesting thing is to create a NUCA-aware allocator that takes the differences between accessing different cache-lines into account. Also, a new version of the Xeon Phi (KNL) was released not so long ago. There might appear some new issues that are specific for this hardware.

Attachment

- **bobox_results –** Results of testing on Bobox
- **mcm** – Results of latencies between memory controllers and cores
- **tdl** – Result of latencies between cache lines and cores
- **bobox_boxes.hpp** – A header file for the schedulable broadcast box
- **boxes.cpp –** A source file for the schedulable broadcast box
- **how_to_build_kmod.txt** – Instructions how to build a kernel module for Xeon Phi
- **Makefile** – Makefile for kernel modules
- **mcm.c** – Source code of mcm measurement
- **tdl.c** – Source code of tdl measurement

Bibliography

[1] D. Bednárek, J. Dokulil, J. Yaghob and F. Zavoral, "Bobox: Parallelization Framework for Data Processing," *Advances in Information Technology and Applied Computing,* no. 2251-3418, pp. 189-194,, 2012.

[2] Z. Falt, "Towards Efficient Parallel Data Processing on Modern Hardware," 2014.

[3] J. Fang, A. L. Varbanescu, H. Sips, L. Zhang, Y. Che and C. Xu, "An Empirical Study of Intel Xeon Phi," China, 2013.

[4] Intel, "Intel Xeon Phi Coprocessor Vector Microarchitecture," 2012.

[5] C. Zhang, L. Li, L. Ruizhe and G. Yang, "Performance Characterization and Optimization for Intel Xeon Phi Coprocessor," 2015.

[6] Intel, "Intel VTune Amplifier".

[7] Z. Falt, "Plánovač a paměťový alokátor pro systém Bobox," 2010.

[8] Intel, "Intel Xeon Phi Coprocessor System Software Developers Guide," 2014.

[9] Intel, "Pentium Processor Family Developer's Manual, Volume 3: Architecture and Programming Manual," 1996.

[10] Intel, "Intel Xeon Phi Core Micro-architecture," 2013.

[11] Intel, "Intel Xeon Phi x100 Family Coprocessor - the Architecture," 2012. [Online]. Available: https://software.intel.com/en-us/articles/intel-xeon-phi-coprocessor-codename-knights-corner.

[12] "Busybox," [Online]. Available: https://busybox.net/.

[13] Intel, "Intel Manycore Platform Software Stack (Intel MPSS)," 2016.

[14] P. J. Salzman, M. Burian and O. Pomerantz, "The Linux Kernel Module Programming Guide," [Online]. Available: http://tldp.org/LDP/lkmpg/2.6/html/lkmpg.html.

[15] J. McCalpin, "How Xeon Phi divides address space with distributed L2 (forum)," [Online]. Available: https://software.intel.com/en-us/forums/intel-many-integrated-core/topic/586138.

[16] J. Li, R. Rajamony, W. Speght, X. Wu and Z. Lixin, "Non-uniform cache architecture (NUCA)". Patent US9152569 B2, 6 October 2015.

[17] M. I. Gordon, W. Thies and S. Amarasinghe, "Exploiting coarse-grained task, data, and pipeline parallelism in stream programs," *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems,* pp. 151-162, 2006.

[18] Z. Falt, M. Čermák, J. Dokulil and F. Zavoral, "Parallel SPARQL Query Processing Using Bobox," *Internation Journal On Advances in Intelligent Systems,* vol. 5, pp. 302-314, 2012.

[19] G. Klyne and J. J. Carroll, "Resource description framework (RDF): Concepts and abstract syntax," 2006.

[20] M. Schmidt, T. Hornung, G. Lausen and C. Pinkel, "SP2Bench: a SPARQL performnace benchmark," *Data Engineering, IEEE 25th International Conference,* pp. 222-223, 2009.

[21] Intel, "How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Achitectures," 2010.

List of Abbreviations

- **SMP –** Symmetric Multiprocessing
- **NUMA –** Non-Uniform memory access
- **VPU** – Vector Processor Unit
- **HT** – Hyper-Threading technology
- **CRI** – Core Ring Interface
- **ICache** – Instruction cache
- **DCache** – Data cache
- **TD** – Tag Directory
- **DTD** – Distributed Tag Directory
- **RS** – Ring Stop
- **APIC** – Asynchronous Interrupt Controller
- **RAM** – Random Access Memory
- **DMA** – Direct Memory Access
- **TDL** – Translation Lookaside Buffer
- **PF** – Picker Function
- **PPF** – Pre Picker Function
- **ECC** – Error Correcting Code
- **FSB** – Front Side Bus
- **LRU** – Least Recently Used
- **ALU** – Arithmetic Logic Unit
- **UALU** – Arithmetic Logic Unit in VPU
- **MESI –** Modified Exclusive Shared Invalid protocol
- **MOESI –** Modifier Owned Exclusive Shared Invalid protocol
- **PCI –** Peripheral Component Interconnect
- **I/O –** Input/Output
- **API –** Application Programming Interface
- **HPC –** High-Performance Computing
- **TCP –** Transmission Control Protocol
- **IP –** Internet Protocol
- **UDP –** User Datagram Protocol
- **MPI –** Message Passing Interface
- **OpenCL –** Open Computing Language

- **OpenMP –** Open Multi-Processing
- **SCIF –** Symmetric Communication Interface
- **MPSS –** Many-core Platform Software Stack
- **FS –** File System
- **NFS –** Network File System
- **GCC –** The GNU Compiler Collection
- **CPU -** Central Processing Unit
- **NUCA –** Non-uniform Cache Architecture
- **ID –**Identification
- **SSH –** Secure Shell
- **SCP –** Secure Copy
- **GBOX –** Memory Controller of Xeon Phi
- **SBOX -** PCI Express Client Logic Used in Xeon Phi
- **DBOX –** Display Engine Used in Xeon Phi
- **GOLS3 –** A Protocol that mimics O state for the MESI protocol
- **DSL –** Domain Specific Language
- **GPL –** General Purpose Language
- **RDF –** The Resource Description Framework
- **SPARQL –** SPARQL Protocol and RDF Query Language
- **KNC –** Xeon Phi Knights Corner
- **KNL –** Xeon Phi Knights Landing