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 well known problem on traditional system architectures (NUMA, SMP). However, it does not perform well on emerging many-core architectures, especially on Intel Xeon Phi. We have collected all publicly available information about the Xeon Phi’s architecture. After that we have benchmarked the Xeon Phi to find missing information about its architecture. We were especially curious in architecture of cores and memory controllers. These parts are most important while designing scheduler. Based on the measured results we have proposed improvements to scheduling algorithm in the Bobox - experimental streaming system.

Keywords: scheduling, streaming systems, Bobox, Xeon Phi
First of all, I would like to thank to my supervisor RNDr. Jakub Yaghob, Ph.D. for his patience, thoughtful advice, and mostly for his time. Another thanks go to my family for their love, trust and that they always stand by me. Last but not least, I thank my friends Miloš Chromý and Jakub Šťasta who proofread my thesis.
## Contents

**Introduction** .................................................................................................................................................. 1

1. **Xeon Phi overview** .................................................................................................................................. 2  
   1.1. **Hardware architecture** ................................................................................................................. 2  
      1.1.1. **Core** ....................................................................................................................................... 3  
      1.1.2. **Memory** ................................................................................................................................. 10  
      1.1.3. **Ring interconnect** .................................................................................................................. 17  
   1.2. **Software architecture** ................................................................................................................... 18  
      1.2.1. **MPSS** ..................................................................................................................................... 20  
      1.2.2. **Overlay system** ..................................................................................................................... 21  

2. **Preparation** ............................................................................................................................................. 22

3. **Testing** ..................................................................................................................................................... 23  
   3.1. **Methodology** ................................................................................................................................... 23  
   3.2. **Testing architecture layout** ......................................................................................................... 25  
      3.2.1. **Memory controllers** ............................................................................................................... 25  
      3.2.2. **DTDs** ..................................................................................................................................... 36  

4. **Application** ............................................................................................................................................. 42  
   4.1. **Bobox** ............................................................................................................................................ 42  
      4.1.1. **Task scheduler** ...................................................................................................................... 42  
      4.1.2. **Memory allocator** .................................................................................................................. 43  
   4.2. **Proposed improvements** .............................................................................................................. 44  
      4.2.1. **Bobox improvements** .......................................................................................................... 46  

5. **Conclusion** ............................................................................................................................................. 47  
   5.1. **Future work** ................................................................................................................................... 47

**Attachment** .................................................................................................................................................. 48

**Bibliography** .................................................................................................................................................. 49

**List of Abbreviations** .................................................................................................................................. 50
Introduction

There have been always effort to be able execute programs as fast as possible. That apply especially for applications processing data. There are multiple ways how to speed up a microprocessor. At first main speedup was done by raising of coprocessor’s working frequency. Few years ago processors with multiple core inside emerged. Processors have started being connected to each other and new architectures have appeared. Intel has come with its many-core architecture and launched Xeon Phi cards.

While trying to run an experimental streaming system, Bobox, on Xeon Phi, it showed that it did not perform well. It seemed there might be a performance problem with a scheduler, however there is no problem with traditional system architectures, i.e. SMP and NUMA. It seemed that we are missing important architecture details that affect the speed of the application.

In this thesis, we want to find the cause of the performance problems on the Xeon Phi. First, we make a summary of all available information about Xeon’s Phi architecture. Then we benchmark the coprocessor to analyze the results to find additional information about its architecture. These information are used as a proposal of possible improvements to Bobox’s scheduler and memory allocator.

First chapter makes overview about Xeon’s Phi hardware and software architecture. It is a summary from all public documents we were able to find. Hardware architecture is thorough and goes in depth in things connected to core and memory layout. These components are important when dealing with scheduling. On the other hand, software architecture contains only basic information necessary for Xeon Phi usage. Second chapter describes preparation of Xeon Phi, its environment and describes some troubles we had while setting things up. Third chapter is about testing. It starts with description of used methodology and continues with results of our testing. Based on these tests, we have recognized and confirmed missing information about Xeon’s Phi hardware architecture. Fourth chapter describes the streaming system Bobox, its internals and possible improvements. Last chapter summarizes our thesis.
1. 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 second, with attribute Phi in its name, is compounded from many small processors and its architecture is closer to architecture used in graphics card.

The latest architecture in the time of writing this thesis is Knights Landing (launched in June 2016). The second latest architecture is called Knights Corner. Because we have a card only with Knights Corner architecture at our disposal, we are not interested in the other ones. Therefore, all things mentioned here are related to this architecture, if not said otherwise.

There are 8 types of Intel Xeon Phi coprocessor with architecture Knights Corner. The only difference is in number of cores, processor base frequency, and number of memory channels and maximum size of used memory. For further simplification we write this thesis only about the model Intel Xeon Phi 7120.

1.1. 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 interconnect. Each core contains: 512-bit vector processor unit (VPU), the core ring interface (CRI), the interfaces to the core and the ring interconnect, the L2 cache, the tag directory (TD) and asynchronous processor interrupt controller (APIC). Other stuff like memory controller (GBOX) and PCI Express client logic (SBOX) are placed on interconnect.

All above components are shown in Figure 1. Core fetches and decodes instructions from hardware threads. Vector processor unit executes floating-point and integer operations. The core-ring interface hosts L2 cache and connects each core to ring stop. APIC and TD are also placed in the CRI. TD’s function is to hold coherence between core’s L2 caches. Memory controller comprises 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 power management capabilities. All components are placed on the ring interconnect.
1.1.1. Core

Xeon Phi core was designed based on old Pentium P54c architecture, but many changes were made. Each core supports fetch and decode instructions from 4 hardware thread contexts that support 32-bit and 64-bit execution environment. It contains 8-way 32KB L1 Icache and Dcache and interfaces with L2/CRI block. Each thread has replicated architecture state – registers: GPRs, ST0-ST7, segment registers, CR, DR, EFLAGS and other states like the prefetch buffers, the instruction pointers, the segment descriptors and the exception logic. Other adjustments to support hardware threads needed to be introduced i.e. ID bits in TLBs, thread-specific flush and hardware support for thread wakeup/sleep mechanisms.
Pipelines

The core pipeline is composed of 7 stages. Pipeline for vector instructions adds another 6 additional stages. Figure 2 shows the core pipeline. Figure 3 shows the vector pipeline, where orange boxes show additional stages in vector pipeline compared to core pipeline. Both pipelines use global stall architecture, which means that part of the pipeline would stall if a stage is being stalled. Some buffers are placed between frontend and backend to queue the stalls.

![Core pipeline](image)

**Figure 2: Core pipeline**

![Vector pipeline](image)

**Figure 3: Vector pipeline**

![Core pipeline compared to the generic one](image)

**Figure 4: Core pipeline compared to the generic one**

Core pipeline

Each core stage is speculative, that means if a branch misprediction or a cache or TLB miss appears, it may invalidate whole work in pipeline. Machine state is updated in the WB stage where all results are written to the machine. Figure 4 depicts the Xeon Phi’s core pipeline compared to the generic one. Instruction fetch is split into two stages – PPF and PF. Purpose of those stages is to choose the thread that will be executed. The PPF stage prefetches instructions for a thread context into the prefetch buffers. There are 4 prefetch buffers per thread and each can contain 32 bytes in a buffer. There are also two instruction streams. If one of the streams is stalled due to a branch misprediction, the second stream is switched in while the branched target stream is being prefetched. Then the PF (picker function) examines the prefetch buffer to determine the next thread. Priority to refill a prefetch buffer is given to a thread
executed in current cycle. Each core contains ready-to-run buffer consisted 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. If the instruction cache does not have control transfer point, a core stall happens. To hide a performance penalty caused by a core stall, the PF chooses next context to execute in a round-robin fashion. If instructions from context 0 are processed in cycle N then the PF will try to issue instructions from context 1, context 2, or from context 3 in cycle N+1. The PF tries to choose exactly in that order.

Two-cycle instruction fetcher unit implies that fully pipelined core cannot issue instructions from the same context in back-to-back cycles. Meaning, if instructions from context 0 are issued in cycle N, then in next cycle N+1, instructions from all contexts except context 0 can be issued. This implies that for full core utilization, at least 2 threads need to be present. Otherwise, thanks to not being able to issue instructions in back-to-back cycle, at most 50% core utilization can be achieved. The refill of instruction buffer that happens during a core stall takes 4-5 clock cycles, which implies that 3-4 threads are needed for optimal performance it this case. On the other hand, if the PPF and the PF are perfectly synchronized, two threads are sufficient for running maximum speed. When they are not synchronized due a cache miss, 1 clock cycle bubble might be inserted. Nice overview of first two stages is shown in Figure 5. Another view on instruction and data flow is depicted in Figure 6.

![Figure 5: Core multithreading support (source [1])]
After choosing instructions by PF, stages D0 and D1 decode them in two clock cycle. Instruction prefixes are decoded at D0 stage. Fast prefixes are decoded without penalty, legacy prefixes have 2 clock cycle latency. In the D1 stage, the instruction microcode found in the ucode ROM is mixed together with the ucode that is generated by the D0 stage. In next decoding stage (D2), the coprocessor reads the general purpose register file and after some computation, it looks up the data cache. The decoded instructions are sent to execution unit by two pipelines – U-pipe and V-pipe. Names are the same like in old Pentium. First instruction takes U-pipe and the second in the pair takes the V-pipe, if it is pairable with instruction in U-pipe. Concurrent execution of instruction is governed by pairing rules. Intel documentation [2] 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.” Lots of other documents about Xeon Phi simply takes over the sentence. Few other documents are using other Intel document [1] that 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 other source implies that not all vector instructions are forbidden to
run on V-pipe. The document also gives a list of instructions that may run on V-pipe. These instructions are shown in Table 1.

![Table 1: Pairable instructions]

At this stage, integer instructions are executed in the ALUs. Once the scalar instructions reached WB stage, they are finished. All core stages with its functions are written in Table 2.

![Table 2: Core pipeline stages functions]

**Vector pipeline**

There is separate pipeline for x87 floating point instructions and vector instructions that start right after the core pipeline. Even though the vector instruction reaches WB stages, it is not finished. The core pipeline may think so, but the vector
unit keeps working until it goes through the whole vector pipeline. At this stage cannot be raised any exceptions. VXCSR flag is set instead to indicate exception condition i.e. underflow and overflow.

Vector processing unit (VPU)

There is no support for extended or vector operation known from other Intel processors – no MMX, Intel AVX or Intel SSE. New vector floating-point unit (VPU) in each core were introduced. It is 512-bit wide and the need of new instructions was necessary. It could execute most instructions in 4 clock cycle with 1 cycle throughput. It can read/write one vector to/from memory. A vector can include either sixteen 32-bit numbers – integer, single-precision floating point or eight 64-bit number – 64-bit integer, double-precision floating point per cycle. Each VPU has 8 UALUs each containing 2 SP (single precision) and 1 DP (double precision) ALUs with independent pipelines. Each VPU has 128 entry vector registers divided equally among all 4 threads making 32 entries per thread. There are additional eight 16-bit mask registers controlling which elements are active during computation. According to [3], each VPU instruction has to pass through all five pipelines to completion:

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

Transition between pipelines costs sometimes additional clock cycles. Some pipelines have built-in bypasses, e.g. SP ALU, so execution of SP instructions consecutively results in good performance. Opposite to code, where SP and DP instructions are mixed, because there are no bypasses between those two ALUs.

Vector pipeline is shown in Figure 3. Whether the processor deals with a vector instruction is found out in the D2 stage. At the E stage it detects any dependency stalls. At next vector stage – VC1 and VC2, the VPU does the shuffling and load conversions. Main add and multiply operations are done at the V1-V4 stage. Last stage – WB writes the vector/mask register content back to a cache. Detailed description of vector processing unit is depicted in Figure 7.
Figure 7: Vector processing unit (source [4])

Latencies

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 any data dependencies, say in two consecutive SP instructions, the second instruction waits until data is produced by the V4 stage and pass it 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, since it has no data from the previous instruction.

If an SP instruction is followed by another SP instruction with register swizzle, data has to be sent from V4 to VC1 stage by another internal bypass. This causes an additional 5-cycle delay. If an SP instruction is followed by an EMU instruction, since EMU’s transcendental lookup happens in VC1 stage, data can be moved by an internal bypass with 5-cycle latency delay. When a DP instruction is followed by a SP instruction, the DP instruction have to complete write before the dependent SP instruction can execute, since there are no bypasses between SP and DP pipelines. This waiting costs 7 clock cycles.

Vector registers

Each thread has thirty-two 512-bit general vector registers \texttt{zmm0-zmm31}, eight 16-bit mask registers \texttt{K0-K7} and the status register \texttt{VXCSR} at its disposal. Vector registers operate either with sixteen 32-bit elements or eight 64-bit elements. The \texttt{VXCRS} holds the status of each vector operation. The VPU reads and writes a data
cache at cache-line granularity of 512 bits (64 bytes) through dedicated 512-bit bus. The data read from cache goes through load conversion and swizzling to the ALU. The writes go also through conversion and alignment right into the write-commit buffer in the data cache.

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

**Extended Math Unit (EMU)**

The VPU also contains extended math unit (EMU). It executes single-precision transcendental operations using quadratic minimax polynomial approximation and does a table lookup for fast approximation to the approximation function. EMU is fully pipelined. The table lookup happens in parallel. EMU offers these basic transcendental functions: reciprocals, reciprocal square roots, base 2 exponential and base 2 logarithms. Other functions – division, square root and power are composed from the basic transcendental functions. Table 3 shows latencies of EMU instructions.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Latency [cycles]</th>
<th>Throughput [cycles]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exp2</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Log2</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Recip</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Rsqrt</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>Power</td>
<td>16</td>
<td>4</td>
</tr>
<tr>
<td>Sqrt</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>Div</td>
<td>8</td>
<td>2</td>
</tr>
</tbody>
</table>

*Table 3: EMU instructions latencies*

1.1.2. Memory

Figure 8 shows a distribution of the memory controllers. As we can see, each core contains L2 cache. The memory controllers and TDs are symmetrically placed around the bidirectional ring.
L1 cache

As any typical processor, a core contains L1 cache. It has 32KB instruction cache and 32KB data cache. It has standard 64 cache line size with 8-way associativity. Data from the cache can be returned out-of-order. For integer values load-to-use latency is one cycle, but for vector values is higher. The cache also contains an address generation interlock with latency minimum three clock cycle long. Therefore, GPR registers must be produced three or more clocks before being used as a base or an index register in an address computation. The base and the index registers setup time have the same three clock cycle latency.

Figure 8: Interleaved memory access (source [4])
L2 cache

L2 cache is inclusive of the L1 data and instruction caches, meaning that all data that are stored in L1 cache has to be duplicated in L2 cache. Its size is 512KB per core. It is divided into 1024 sets with 8-way associativity per set and 64 bytes per a way. It is split into 2 banks and is able to address 32GB (35 bits). The latency of L2 cache is 11 clock cycles and the idle access time is around 80 clock cycles. Opposite to L1 cache it cannot be accessed each clock cycle, but only every other clock cycle. L2 cache contains streaming hardware prefetcher that is able to prefetch a code, read and read for ownership (RFO) cache lines into cache. There are 16 streams that might bring in up to 4KB page of data. If a direction of stream is detected, up to four multiple prefetch requests could be issued. The cache supports ECC and different core’s power states like C1, C6 and the package C3 states.

L2 cache is part of core ring interface (CRI). CRI contains also tag directory (TD) and ring stop (RS), which connects interconnect. These blocks form Transaction Protocol Engine that interfaces to RS and is equivalent to FSB unit. Ring stop takes care of all traffic that comes on and off the ring. The TD’s function is to filter and to forward requests to appropriate agents on the ring. They also initiate communication to memory via memory controllers.

All cores together contain around 30MB of L2 cache – each core provides its 512KB. This is valid only if data among cores are not shared. Then each core has only its private data. The sharing data are replicated in each core that shares the data. In worst case scenario, in which all data are shared among all cores, effective size of L2 cache is only 512KB.

Both caches

Both caches uses a pseudo-LRU algorithm for replacement and are fully coherent. Both caches support 38 outstanding requests per core (both read and write). The system agent (containing PCI Express agent and the DMA controller) can generate additional 128 requests (read and write). Either read or write to cache can be run in one clock cycle. The total number of maximum requests is $38 \times \text{(number of cores)} + 128$. This allows software to prefetch data aggressively. When the maximum number of requests is reached, new requests cause a core stall.
The standard MESI protocol is used for maintaining shared state across the cores. How the MESI protocol works is shown in Figure 9. Its states are described in Table 4.

![MESI protocol diagram](source [1])

<table>
<thead>
<tr>
<th>Cache state</th>
<th>State definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>M (Modified)</td>
<td>Cacheline is updated relative to memory (GDDR). Only one core may have a given line in M-state at a time.</td>
</tr>
<tr>
<td>E (Exclusive)</td>
<td>Cacheline is consistent with memory. Only one core may have a given line in E-state at a time.</td>
</tr>
<tr>
<td>S (Shared)</td>
<td>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.</td>
</tr>
<tr>
<td>I (Invalid)</td>
<td>Cacheline is not present in this core cache.</td>
</tr>
</tbody>
</table>

Table 4: Cache states
Tag directory (TD)

To solve potential performance limitations resulting from missing O (owner) state found in MOESI protocol, Xeon Phi presents an ownership tag directory (TD) similar to that implemented in some multiprocessor systems. MOESI protocol avoids the need to write back modified data to main memory before sharing it. Because TD is faster than the memory, we should gain the promised performance boost compared to MESI protocol.

Table 5 describes MOESI protocol. By supplementing individual core’s MESI protocol with the TD, it is possible to emulate O state in the MOESI protocol without the need of redesigning core’s cache blocks. The modified coherence diagram for MESI and tag directory’s GOLS3 protocol together is depicted in Figure 11. Tag directory implements GOLS3 protocol. GOLS3 protocol state diagram is shown in Figure 10 and each state function is described in Table 6.

<table>
<thead>
<tr>
<th>Cache state</th>
<th>State definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>M (Modified)</td>
<td>The cache has only valid copy of the cache line and made some changes to the copy</td>
</tr>
<tr>
<td>O (Owned)</td>
<td>The cache is one of many with a valid copy of the cache line and has exclusive right to modify it.</td>
</tr>
<tr>
<td>E (Exclusive)</td>
<td>The cache has the only copy of the cache line and the cache line is unmodified</td>
</tr>
<tr>
<td>S (Shared)</td>
<td>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.</td>
</tr>
<tr>
<td>I (Invalid)</td>
<td>The cache line is not valid. Must be fetched from memory.</td>
</tr>
</tbody>
</table>

Table 5: MOESI protocol
Figure 10: GOLS protocol in the TD (source [1])

<table>
<thead>
<tr>
<th>TD State</th>
<th>State definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOLS (Globally owned, locally shared)</td>
<td>Cacheline is present in one or more cores, but is not consistent with memory.</td>
</tr>
<tr>
<td>GS (Globally shared)</td>
<td>Cacheline is present in one or more cores and consistent with memory.</td>
</tr>
<tr>
<td>GE/GM (Globally exclusive/modified)</td>
<td>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.</td>
</tr>
<tr>
<td>GI (Globally invalid)</td>
<td>Cacheline is not present in any core.</td>
</tr>
</tbody>
</table>

Table 6: GOLS protocols states
Tag directory (TD) is distributed among cores in 64 smaller distributed tag directories (DTDs). Each TD takes care about the global coherence state for its assigned cache lines. It contains the address, GOLS3 state and an ID for the owner of cache line. Cache lines are uniformly distributed amongst the tag directories. That should provide a uniform traffic characteristic on the ring.

Information about cache line state is associated with each cache line and is stored in individual cores. Cache lines state uses standard MESI protocol. If a core find out, while a cache line load, that the cache line is in modified state, then it sends a request to DTDs and gets cache line’s GOLS3 state. Those information together emulate MOESI protocol.

**Page tables**

Two types of memory are used in Xeon Phi – uncacheable (UC) and write-back (WB). L1 data TLB supports three types of page sizes – 4KB, 64KB and 2MB. It is able to address 32-bit physical addresses, 36-bit physical addresses extension (PAE) in 32-bit mode and 40-bit addresses in 64-bit mode. The execute disable (NX) bit is supported, in opposite to global page (GP) bit that is presented on other Intel microprocessors.

If a request misses L1 and also L2 TLB, 4-level page table walk is performed. There are no restrictions for mixing page sizes in a single address block, but they all
have to be consistent. The Table 7 shows characteristics for TLBs located on Xeon Phi. All TLBs are 4-way.

Instruction cache does not support large pages. TLBs can share entries among threads that have same values as TLB’s registers: CR3, CR0.PG, CR4.PAE, CR4.PSE, EFER.LMA.

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Entries</th>
<th>Maps</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Data TLB</td>
<td>4KB</td>
<td>64</td>
<td>256KB</td>
</tr>
<tr>
<td></td>
<td>64KB</td>
<td>32</td>
<td>2MB</td>
</tr>
<tr>
<td></td>
<td>2MB</td>
<td>8</td>
<td>16MB</td>
</tr>
<tr>
<td>L1 Instruction TLB</td>
<td>4KB</td>
<td>32</td>
<td>128KB</td>
</tr>
<tr>
<td>L2 TLB</td>
<td>4KB, 64KB, 2MB</td>
<td>64</td>
<td>128MB</td>
</tr>
</tbody>
</table>

Table 7: TLB’s characteristics

Memory controller

There are 8 on-die GDDR5 memory controllers. Each can operate two 32-bit channels for a total of 16 memory channels that can deliver data 5.5 GT/s per channel. The controllers directly interface with the ring. They are responsible for reading and writing data to GDDR memory by translating memory reads and writes to commands. The bandwidth guaranteed to the SBOX is 2GB/s. It supports the ECC data integrity feature.

1.1.3. Ring interconnect

Ring interconnect is implemented as a bidirectional ring. It contains three independent rings in each direction. Details are shown on Figure 12. The largest, 64 bytes wide, is data block ring. The address ring is much smaller and is used to send read/write commands and memory addresses. The smallest and the fastest ring is a acknowledgement ring. It is used to send flow control and coherence messages.
Whenever an L2 miss appears on a core, the core generates a request on the address ring and queries tag directory. If requested data are found in L2 cache, a forwarding request is sent to that cache over the address ring and the requested block is sent back through the data block ring. If the address is not found in any tag directories, appropriate tag directory generates a memory address request to the memory controllers. After that, data are returned 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 higher number of requests on the ring.

1.2. Software architecture

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

Each card represents Symmetric Multiprocessing (SMP) domain that is loosely coupled to the computing domain represented by a host platform.

Lots of APIs are provided to support many High-Performance Computing (HPC) applications. There are TCP/IP, MPI, OpenCL APIs and some Intel interfaces to create a suitable abstraction layer. The communication layer between a host and the card is called SCIF. Figure 13 shows relationships between APIs and other
components that altogether create Manycore Platform Software Stack (MPSS). MPSS is collective name for all software on the host and the card that supports the coprocessor. The left side of picture shows the host stack that runs on standard Linux kernel, the right side shows similar stack that belongs to the coprocessor and runs on top of Linux based kernel with Xeon Phi specifications.

Figure 13: Xeon Phi software architecture (source [1])

The software architecture supports three programming models – offload, symmetric and native. A main application is launched on host processor in the offload model. An offload process is controlled by offload pragmas and offloading is fully in hands of a compiler. The symmetric model is composed of multiple processes, where their computation and communication is not done automatically by a compiler, but have to be done explicitly using some standard mechanism e.g. MPI. The last native model is a variant in which an application is running exclusively on Xeon Phi coprocessor.
The Xeon Phi card does not contain a permanent file system (FS) storage, hence the file system is maintained in RAM and can be remotely mounted e.g. through NFS.

1.2.1. MPSS

Figure 14 pictures a high level representation of the MPSS. Although the stacks contain similar components, they are not binary compatible, because they both run on different platforms. Logically, the same thing holds for applications – native Xeon Phi applications are not binary compatible with any other applications.

![Figure 14: The MPSS (source [1])](image)

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

A Linux utilities are provided by BusyBox. These utilities provide a basic functionality compared to Linux host distribution.
Virtual Ethernet drivers implements a virtual Ethernet transport between host and coprocessor. There is support for TCP, UDP, IP stack and tools such SSH, SCP etc.

SCIF is a low-level communication layer between host and coprocessors. It provides an API for communication between host and coprocessor and also between multiple coprocessors within a platform. It exposes DMA capability for transfers and ability to map memory from coprocessor to host and in the opposite direction. SCIF is built on top of PCI Express which results in the advantage of inherent reliability and no need to check any data packages for error opposite to Virtual Ethernet driver.

1.2.2. Overlay system

There are prepared some system administration tools for Xeon Phi on host. They serve to control a coprocessor. Second option to control a coprocessor is through a configuration files. They are located in /etc/sysconfig/mic/ on host. The directory includes a file for the configuration of all coprocessor installed on the the system – default.conf, configuration files for each installed coprocessor – mic<n>.conf (where <n> is a coprocessor number) and configuration files for configuration of software we want to install – conf.d/*. Card’s kernel module is configured using the file /etc/modprobe.d/mic.conf on host. Every change to these files requires a coprocessor restart for changes to take an effect.

The root file system for coprocessor is constructed on host and is compressed into a cpio file. On boot, content of the file is unpacked and loaded into RAM disk. The file system image is not directly modified, instead one or more file hierarchies overlay corresponding files in the file system image during a boot. That means, each file in a hierarchy replaces corresponding file in the base file system or creates it if it does not exist in the base file system. These hierarchies are called overlay sets or overlays. 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 every coprocessor.

The overlay process begins with the Base file system whose configuration specifies the file system to be used. The common overlay can be populated with files that overlay the file system of all coprocessors. It is rooted at /var/mpss/common by default. There also exists per-coprocessor overlay that enables overlaying of files per coprocessor. This overlay includes some files by default (see [5] for more info).
2. Preparation

Before starting, we need to prepare a testing environment. Even though the coprocessor uses a Linux kernel, it has its specifics. Some of these specifics are not described in any manuals and were discovered on the fly. Also there are not many sources on the internet, owing to the fact that user community around the coprocessor is not so large.

Compiling kernel module

Building a kernel module for Xeon Phi is not so different from building a kernel module for other platforms. The infrastructure to building kernel and managing all compilation options is known as Kernel Build System – kbuild. Our Xeon Phi runs kernel in version 2.6.38 with MPSS 3.7. While there are newer versions of Linux kernels, there is available only this old version kernel on Xeon Phi. On one hand there are no features from newer kernels, on the other hand the kernel is stable.

The kernel was built on host, because there are not any required utilities on the coprocessor, at least not on our one. This leads to a problem that we had to cross compile the kernel module. First, we got source codes of our kernel. It might be downloaded or accessed through e.g. NFS from a coprocessor. Secondly, we used a compiler that is able to create binary files for Xeon Phi - there exists a version of GCC specially prepared to be able to produce Xeon Phi’s binaries. A path to GCC has to be provided to kbuild. We simply added the path to PATH variable. Last thing that cannot be forgotten is to specify the built architecture to kbuild. The architecture should correspond to build tools that we added on PATH.

Root privileges on coprocessor

Loading a kernel module can be done in two ways. First possibility is to load it while booting coprocessor. A kernel module has to be defined in the overlays. Disadvantage is a necessity to reboot coprocessor every time a change was done to a kernel module. Easier way to handle loading and unloading of a kernel module is with standard utilities insmod and rmmod. To be able to call the utilities, we wanted to gain root privileges on coprocessor.

Straight process on standard Linux would be to add entry into /etc/sudoers. We cannot modify the file directly because changes would last only until next reboot. We had to use overlay to preserve changes. We tried to simply add the file into an
overlay definition, but it did not work. There is nothing about this special behavior in any documentation. There is, however, a note that some files are included by default into a per-processor overlay. Almost all special files from /etc directory are included, but not sudoers. It seems that the files from /etc are either included by default into an overlay or cannot be included at all.

And yet, it is possible to add user to sudoers. Luckily, there is entry in default sudoers which includes additional directories (sudoers.d) where any additional rules can be added. Our final solution to add user into sudoers was to create a file with desired sudoers rule and add a definition into an overlay saying the created file will be copied into directory /etc/sudoers.d.

3. Testing

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

3.1. Methodology

We needed to measure time spent solely in our code. At first we tried to measure time from user space, but we found out, that to be able to access kernel structures and functions, it would be necessary to write a kernel module. At that point we decided to measure time reading a timestamp counter, because it gives us more accurate results than other time utilities.

RDTSC and CPUID

All Intel CPUs have monitoring counter to keep track of events that happens inside a processor. One of the counters is a timestamp counter that counts every cycle that happens in a processor. This counter can be accessed by RDTSC or RDTSCP assembly instructions. RDTSC was introduced by the Pentium processor. RDTSCP was introduced later and is not always supported by all processors, especially Xeon Phi does support only the RDTSC version.

Both instructions read timestamp counter (a 64-bit MSR) into EDX:EAX registers. EDX contains the high-order 32 bits of MSR and EAX the low-order 32 bits. RDTSC is not serializing instruction. It does not wait until previous instructions have been executed before reading the counter. That differs to the RDTSCP which waits.
Subsequent instructions might begin execution before read operation begins, that is similar for both instructions. To be able measure only instructions we desire, some kind of fence needs to be added before and after \texttt{RDTSC} instruction.

Intel architecture defines several serializing instructions. These instructions force processor to complete all modifications to flags, registers and memory by previous instructions and to drain all buffered writes to memory before the next instruction is fetched and executed. This means that nothing can pass a serializing instruction and a serializing instruction cannot pass any other instruction.

One of the instructions that could be used for serialization is \texttt{CPUID}. Its main purpose is to return processor identification and its features. It also guarantees that any modifications to flags, registers and memory for previous instructions are completed before the next instruction is fetched and executed.

With right combination of instructions above, we are able to achieve that we measure only the piece of code and no other unwanted instructions. At the beginning, we need to place \texttt{CPUID} instruction before \texttt{RDTSC}. We know that \texttt{CPUID} serialize instructions, meaning that all instructions are executed before. Also \texttt{CPUID} cannot pass through the next instruction which is \texttt{RDTSC}. At the end, \texttt{CPUID} is placed also before \texttt{RDTSC} instruction and \texttt{CPUID} ensures that any instructions below and above \texttt{CPUID} does not start executing before \texttt{RDTSC} reads the counter. One disadvantage is that we measure also a latency of \texttt{CPUID}. If we use \texttt{RDTSCP}, we could insert \texttt{CPUID} after that instruction at the end, because \texttt{RDTSCP} is able to serialize instructions that start before. That would solve the problem with measuring the latency of \texttt{CPUID}.

Described process for measuring is general and could be used for any out-of-order processor with instruction pipelining. Measurement of Xeon Phi is little bit easier, because all coprocessor’s cores are in-order. That means, the coprocessor cannot internally reorder instructions. Still, there is support for pipelining in the coprocessor, so a serializing instruction makes sense. It forces any instructions in any pipeline stage to finish before \texttt{RDTSC} instruction starts to measure.

\textbf{Disabling kernel preemption}

We secured that there is no distraction from the code we wrote. Still, there are other factors that might ruin our measurement. The coprocessor’s OS Linux is preemptive. If its scheduler decides, it might temporarily interrupt running thread and
let another thread to run instead. In our test scenario is it quite unlikely to happen, because we run mostly few threads, but system might want to do its internal stuff on our core. Other things, that might also ruin the measurement, are interrupts.

Our goal is to make sure we exclusively own the CPU the measured code runs on. We call \texttt{preempt\_disable()} for disabling preemption and \texttt{raw\_local\_irq\_save(flags)} for disabling hard interrupts. Pair functions that enables both things are \texttt{raw\_local\_irq\_restore(flags)} and \texttt{preempt\_enable()}.

3.2. Testing architecture layout

We know from all architecture pictures of Xeon Phi that the cores and the memory controllers are placed on a 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. It also seems that the memory controllers are paired. We would try to test, whether access from some CPUs to some controllers takes longer than to the others. It would mean that the system has a NUMA architecture. Next we test access to DTDs from different cores.

3.2.1. Memory controllers

First, we wanted to test latency of access from a core to a memory controller. The architecture picture shows that cores are uniformly distributed among the bidirectional ring. It also shows the memory controllers are paired which would leave us with four groups of memory controllers (there are 8 memory controllers). It would make sense to place those groups uniformly between individual cores. Because Xeon Phi has around 60 cores it implies there has to be around 15 cores between adjacent memory controllers. If we take opposite memory controllers and take closer look on a core that is the closest to the first controller, it would mean there are around 30 cores between the core and the second controller. This should be the longest path between core and memory controller, because the ring is bidirectional and it uses the shortest path between every two elements on the ring. We think, there is insignificant difference between access time to the memory controller from the nearest and the furthest core.
Physical addresses to memory controllers mapping

First, we need to know how addresses are mapped to memory controllers. This information could be read from performance counters inside each memory controller. Where the performance counters are is not public information and it cannot be found in any public documents published by Intel. If we knew where the performance counters are, we would try to read and write a cache-lines one by one while looking into performance counters. It does not make sense that blocks of memory in each controller would be smaller than a cache line. We expect that each controller contains blocks around 4KB which corresponds to size of the smallest virtual pages on a coprocessor. That makes 64 cache lines in each block, because the size of cache line is 64B.

We found following layout of memory on Intel’s forum [6]. Physical memory of coprocessor is divided into blocks of 62 cache lines. The blocks are interleaved around the 8 memory controllers and 2 channels per controller using a fixed repeating pattern. The left 2 cache line in each 4KB range are used for holding ECC data. Figure 15 shows suggested layout. They also propose mapping between memory controllers and physical memory address they got from measuring performance counters. The mapping is written in Table 8. We programmed our tests with belief in given layout and mapping. When we get the results from our measurement, we check characteristics of given results and that should tell us, whether we were right or wrong.

Implementation

For measuring latencies of a memory controller, we need to access only blocks that belongs to it. We used a technique called pointer-chasing. We created a big array of unsigned integers. Each of them is pointing to next element in the array. This creates a chain from pointers. Our program is then traversing the array using read element as an index to the next element. Because we use a read value as the index, prefetcher is not able to do anything with this code. It will not load pieces of memory in advance, so we can see whole latency of reading from memory.
Figure 15: Memory controller mapping

<table>
<thead>
<tr>
<th>Select</th>
<th>Memory Controller</th>
<th>Channel</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>A</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>B</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>D</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>F</td>
<td>7</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 8: Physical address to memory controller mapping
Pseudocode below shows how a traversal code may look like. An array in example is called \( A \) and \( i \) is an index into the array. We did not use this code in our testing program, we manually unrolled the cycle. Not optimized cycles are replaced with comparisons and jumps in assembly code and we wanted to avoid it. We wanted to measure an access to memory only.

\[
i = 0; \\
\text{while}(i < \text{A.size()}) \\
\quad i = \text{A}[i];
\]

We want to create an array with a chain of pointers that would jump only on blocks stored in given memory controller. We know that cache line is 64B long, each block has 62 of them and these blocks are repeatedly distributed among memory controllers. If we want to jump only on first cache line in each controller’s block we have to jump over \( 64B \times 62 \times 8 = 31744B \). If we skip controller’s channels, we have to multiply given number by two.

We allocated \( 2^{10} \) of 4KB pages which is together 4MB of a continuous space. We casted it into an array of `uint64_t`. A length of pointer chain between blocks could be the size of the array divided by step which gives us number lesser than 64.

\[
4MB / (64 \times 62 \times 16) \leq 4MB / 2^{16} = 64.
\]

So we tested pointer chains in length of 64 elements.

We repeated the test many time. In that case we have to take care of caches. If we run the test for the first time, the values are loaded from a main memory – memory controller. While loading the values, data are stored into caches. We would get bad results every other run, if we do not erase the values from caches. Xeon’s Phi caches are using a modification of LRU algorithm to change lines. We had to replace the old values with a new ones. We created an array with size of L2 cache – 512KB. To fill caches with values from the array we simply iterate through the array few times.

We sometimes have to fight with a compiler while writing a test code. The problem is, compilers are quite clever these days and are able to optimize away things that are there for a reason it cannot see. Nice example can be a function for filling the caches with dummy data. If the function went through an array and was doing nothing, it would be most probably eliminated. We added a simple arithmetic (summing of all elements) and then we printed a result.
A memory in kernel can be allocated by two ways. We used `kmalloc()` and `kfree()` for smaller chunks of memory. They are similar to user space `malloc()` and `free()` with difference that `kmalloc()` adds an allocator flag parameter saying how the allocation behaves. For all our kernel allocation we used flag `GFP_KERNEL` that normally allocates a memory in kernel. There is nothing special about this flag. For bigger allocation or allocation of whole pages we used functions `__get_free_pages()` and `free_pages()`. First function allocates contiguous physical pages and needs the allocator flag and number of requested pages. A parameter is an number `order` where a number of allocated pages is calculated as `allocated_pages_count = 2^{order}`. Memory allocated in kernel is un-swappable by default that means that nobody can swap our memory away. That comes in handy because, if we used classic allocators from user space that returns a swappable memory, memory could be relocated and mapped to totally different physical page, which might change associated memory controller. However, there are options how to pin a memory in user space.

For translation of a virtual addresses into a physical addresses, we used a function `virt_to_phys()`. It has one parameter - a virtual address and returns a physical address.

We had pinned memory and we had to do the same with a core. We need to be able to run a single thread on one core and to forbid its scheduling to another core. One way of doing it is through a function `kthread_bind()`. One of its parameter is `task_struct` pointer that holds information about bound thread. Second parameter is a CPU number we want to bind to. The number is number of available threads, not number of cores. For Xeon Phi with 61 core, each having 4 threads, the possible CPU numbers are from 0 to 243. If the function binds, a thread it wants to bind to must be stopped. That is the difference to function `set_cpu_allowed()` that is used for `kthread_bind()` implementation. Another possibility to bind a thread to a CPU is offered by OpenMP. This framework is commonly used in the user space, but it is not suitable for kernel space.

**Results**

We measured a latency between each logical core (a physical core with its threads) and each memory controller with different channels. One measurement cycle does 64 jump in memory allocated on memory controller. Caches are flushed after a
measurement cycle and we measure again. A cycle is repeated 1024 times and an average value is calculated from given values. We repeated this process for each logical core (244) and each memory controller with its channel (16) – 3904 tests together. We could not set a higher number of repeats, because kernel started to time out while loading our kernel module.

Graph 1 shows all taken results. Individual cores are around a graph, each core has an assigned measured value. Further the value from the center of the graph is, the higher cycle latency was measured. The second view on results is depicted in Graph 2.

Every memory controller has its own unique color. 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. Similar results are in [7], where memory read latency from main memory was around 305 cycles. There is difference in order of magnitude compared to 11 cycles of reading from the L2 cache. There is 65 cycles difference between the fastest and the slowest reading from memory. The closer a core to a memory controller is, the faster the reading is. The fastest read should be when the core is next to the controller it reads from. The slowest read is when a core is placed opposite to the memory controller and the fastest core. We could see that in the graph – the highest values are always opposite to the lowest ones. In other graphs, we see a distribution of values is almost linear. Because there are around 30 cores in a half of the ring and the difference is around 60 cycles, it is safe to say, we have to add two clock cycles for each core on the shorter path between the memory controller and the core requesting data. The results are not so bad, but we must recall, only one core was accessing the memory, so there was no use for any synchronization. The second testing program shows how it changes if the data are synchronized through the TDs.

We want to find paired couples of memory controllers in first two pictures. The access latency cannot be the same, but should be similar. We could see the pairs are memory controllers with indices (0, 1), (2, 3), (4, 5) and (6, 7).
Graph 1: Cores' read latency from memory controllers

Graph 2: Cores' read latency from memory controllers
Graph 3, Graph 4, Graph 5 and Graph 6 compare latencies of found memory controller pairs. We added a polynomial trend line to every line to see how the latencies behave. Note that every two trend lines intersect each other exactly two times. One of them is near the minimum and the second is near the maximum. These two intersections correspond to intersections that can be found on the latency lines. They are placed near the extreme values and swaps values behavior. At the beginning, most of the latencies of controller A should be above latencies of controller B. First appearance of extreme changes the ordering – latencies of a controller B are higher than of a controller A. The second extreme changes the ordering back how it was at the beginning.

This behavior tallies to a layout of memory controllers and cores on the ring. If we are getting closer to memory controller pair from one side of the ring, the latencies are falling down, but we are always closer to one of the memory controllers than to another from the pair. The second one is hidden behind the first controller, so a path to it is always a step longer. This changes in the moment we get to the other side of the memory controller pair. The hidden controller starts to be first at the path to this controller pair and we need the extra step to get to the other one. This changes as soon as we get to cores on the opposite side of the ring. At one moment, the shortest path goes in one direction and with next step it starts to go the second direction. This core (or cores) has the longest path to the memory controller which goes hand in hand with the highest latency.

Graph 3: Memory controllers no. 0, no. 1 latencies
Graph 4: Memory controllers no. 2, no. 3 latencies

Graph 5: Memory controllers no. 4, no. 5 latencies
We are able to guess the layout of memory controllers and cores on the ring. Cores, which are the closest to a memory controller, surround from both side an intersection in the picture. The intersection should be near the minimum latency value. Base on this idea, we created Figure 16 of layout of memory controllers and cores on the bidirectional ring.

Graph 7 shows trend lines of latencies for each memory controller. It depicts that memory controllers are placed on the ring and their latencies correspond with this architecture. Moreover, it confirms our hypothesis from the beginning the memory blocks are built from 62 cache lines. If it was not true, the characteristic would not tally to a circle layout. It could be noticed better on the Graph 8 and the Graph 9.
Figure 16: Layout of memory controllers and cores

Graph 7: Trend lines of latencies for each memory controller
3.2.2. DTDs

Second thing we wanted to measure was the influence of DTDs positions on their latencies. There are 64 tag directories (TD) around the ring. One record in TD stores cache lines address, a mask with valid CPUs and a state. It is hard to get information about mapping between cache lines and TDs. There are no performance counters available for TDs, so the only possibility was to find mapping through tests. The only thing we know is that cache lines are uniformly distributed among all TDs.

Our application tests cache line ping-pong between two cores and for different cache lines. Because TDs are uniformly distributed among the ring, we expect it would
take different cores different time to get to different TDs. We were curious about TD to cache line mapping, so there were no need to try all CPUs. We tried few of them and lots of cache lines.

**Implementation**

We created a cache line ping-pong for testing TD’s latencies. We created numerous versions, later we chose the one which had the smallest variance of results.

Our desired application have two threads. First thread accesses cache line (CL) and finds out it is not in its cache. It sends a load request to a TD owning the CL. No other core has the CL in its cache, so the TD sends a request to the memory controller owning the address. Now, the second thread wants to access the CL. It is not in thread’s cache, so it sends a request to the TD. The TD see that the CL is in first thread’s cache, so it sends a request there. Now, the first thread is on its turn. It wants the CL and because the CL is modified, it has to send a request to the TD. The TD sends a request for the CL to the second thread. This repeats and goes on and on. One thread looks into the cache, sends request to the TD, the TD requests the CL from the other thread and the other thread sends data.

If we bind the cores and change only TD, sending data would be the same, because it happens between cores. The only difference in changing the TDs is that the latency of requests would be different. This way we measure the latency of access to different TDs.

First implementation creates false sharing between those two threads. False sharing happens when there are seemingly not related data on the same cache line that are accessed from multiple threads. All caches, memory and tag directory are working with cache lines as their smallest unit. Every change to small piece of cache line implies that the whole cache line has to be invalidated. Our program created structure with two counters, each of them for a different thread. The structure was mapped to the cache line and threads started to increment their counters. We did not add any synchronization and let the threads to run. First, we let the slave thread to run and after its start, we started a thread that measured the code. This implementation shows worse results that the second implementation. We think it is because threads were not forced to be run simultaneously and with exactly same speed. It might happen that a core access the cache line without any interference of the second thread.
The second implementation uses an atomic counter. We used Linux’s type \texttt{atomic\_t}. We did not want to use it at first, because we were afraid of them being slow. It internally uses barriers to synchronize access to memory.

The cache line contains this atomic counter and one thread is incrementing it and the second is decrementing. Meanwhile, one of them is measuring the latency. The results was also not so good. We think the problem is similar to the problem in first implementation. We did not force an exact switching between memory writes.

Our final implementation uses one \texttt{volatile} counter. We used \texttt{volatile} to signal a compiler that it should not try to optimize this variable. Second variable was \texttt{atomic\_t} and it shows whose turn is to play. Both variables were stored on the tested cache line. Waiting for turn was done by classic busy waiting. One catch is hidden in this implementation. We have to remember that we load cache line twice, the first is for the counter value and the second is for turn value.

We allocated 1024 pages and then we tried to play the cache line ping-pong with bound threads. We used \texttt{kthread\_bind()} for binding a thread to core. For creating a thread we used function \texttt{kthread\_create()}. It needs a function to be run in other thread and data it gets as an input parameters and return pointer to \texttt{struct task\_struct}. For starting a thread, we used \texttt{wake\_up\_process()} that starts the function it gets as an input parameter. We used \texttt{kthread\_stop()} and \texttt{kthread\_shoud\_stop()} for ending a thread. The first function sends a signal to thread that it should end and the second function checks whether the signal has arrived.

First, we had a problem to start the threads. Even though we called \texttt{kthread\_stop()} after \texttt{wake\_up\_process()}, some threads did not start. For this reason, we had to create our own signals using \texttt{atomic\_t} that signaled the thread is successfully running. We do not know, why this happened. Either it was due some compiler optimizations or maybe it is some kind of internal behavior.

\textbf{Results}

We tested multiple cores and multiple cache lines. We always bind the cores and then test all cache lines’ latencies in allocated pages. The graphs below are taken examples from whole group of tests. All measured values looked very similar.

One side effect of our testing program is, it is able to find out whether the logical threads are running on the same core. If thread runs on the same core, the state
of cache line is not synchronized through TD, but only through core’s cache. The difference between latency of cache and TD is large.

We found, the IDs used for logical thread are shifted. Each physical core has 4 logical cores (threads). We found that ID == 0 does not belong to core 0, but to core 60. The IDs for each core are group of four, but starting with one. Core 0 has IDs = 1, 2, 3, 4; core 1 has IDs = 5, 6, 7, 8 etc. Finally, core 60 has IDs = 61, 62, 63, 0.

We have measure many times and lots of cache lines, but we were not able to find mapping between cache line address and TD. One interesting thing is, there are same latencies appearing in every page. We think it is because each cache line in every page is mapped to one unique TD. Therefore, all TDs are used in standard 4KB page. If we look at the Graph 10, Graph 11, Graph 12 and Graph 13, we would always found the smallest value (around 288 cycles), the highest value (around 805 cycles) and any other values.

Graph 10: Measuring TD's latency
Graph 11: Measuring TD's latency

Graph 12: Measuring TD's latency
The measured values are normalized. We measured latency of 128 ping-pongs that correspond to 512 memory loads. Each thread is loading 2 cache lines per ping-pong. We are loading twice, because we stored the value of counter and also value of turn on the cache line. We have to load cache line for each of them, therefore we have divided all measured values by 512 to get latency of one memory transfer. We see interesting thing – the fastest latency takes around 144 cycles, opposite to the slowest with around 402 cycles. That is huge difference. The difference is almost in ratio 1:3. The only think that differs between different cache lines is a position of the TD. We still have the same length of the path between cache lines, still the same number of requests to TD.

Even though we do not know what the mapping between TDs and cache line address is, we found that there is huge difference – up to 1:3 in access to TDs, depending on their position.

Another view, if we share data between multiple cores we have to be very careful where it would be stored. Sharing memory almost always implies the use of DTD. There is huge difference – 1:3, on which cache line the shared data are stored. Because we do not know the mapping, the fastest way to find is to test it. We can try to move data to different cache lines and see. We would have to test 64 cache lines at most, because there are 64 of unique latencies in each page and then they start to appear again.
We were talking about sharing small data that fit into one cache line. If the data were bigger, we would start getting closer to average TD’s latency, which is around 260 cycles.

4. Application

We have described information that we have found about architecture of cores and memory controllers in previous sections. Main motivation for looking for deviation and additional information about Xeon Phi architecture was bad performance of task scheduler in Bobox.

4.1. Bobox

Bobox [8] 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 called boxes and their mutual connection known as an execution plan.

Boxes are responsible for processing data. Each of them transform its input streams into output streams. The streams are flow of envelopes. The envelopes are sequences of tuples. The envelope travels from box to box. When it arrives into box, the box is scheduled by a task scheduler. This way the whole execution plan is being evaluated.

The critical parts of the Bobox framework are the task scheduler and the memory allocator. The dissertation thesis [9] contains detailed information about those components. We took and use only information relevant to us.

4.1.1. Task scheduler

Processing unit which share at least one level of cache forms processing group. There exists a thread pool per one processing group. Each pool contains as many threads as there is number of processing units inside the group. Affinity of threads is set to this group. Each processing unit contains one queue of immediate tasks and all processing units inside a processing group share one queue of deferred tasks. Distance of units within one group is the lowest level of cache which these units share. Distance of two groups is the distance of their corresponding NUMA nodes, i.e. equal to the NUMA factor.
The immediate tasks are bound to the processing of input data. It expects to be executed as soon as possible and preferably by a thread which is close to thread which spawned it. Ideal candidate is the thread which spawned it. The deferred task are not bound to any data, so they can be executed by any thread.

When looking for next task, the scheduler selects the closest task to the task executed previously. First, it finds out on which processing units it runs and chooses a task according to first applicable rule below:

1. The youngest task from the queue of immediate tasks of the current processing unit.
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, its 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 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, its oldest task is taken. The immediate queues are scanned in round robin fashion and the thread remembers the last non-empty queue found. When this rule is applied again, the scan is resumed 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 since it may have the context of the tasks created on this unit partially 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

Blocks larger than 512KB are allocated and deallocated using system functions. These blocks should be accessed rarely, because the preferred size of an envelope is always smaller than 256KB.
For blocks between 8KB and 512KB, there exists several fixed-size blocks sub-allocators. When a new block is requested, appropriated sub-allocator, the one with the smallest size of blocks larger or equal to the requested size, is used for returning new blocks. If there is no such block, it allocates super block and uses it as an additional memory pool. The free blocks are stored in a linked list. The allocator uses free blocks in LIFO manner – it returns recently deallocated blocks. Each allocated block has a small header with the information about the sub-allocator and its super block.

The allocation and deallocation of super block is done by the super block allocator. Each NUMA node has one of them and allocates memory preferably from that node.

Blocks smaller than 8KB are allocated similar way. There are also implemented several optimization to reduce the internal fragmentation and overhead of managing small blocks. The need of one global lock for allocation is eliminated by the replication of memory pools.

4.2. Proposed improvements

When starting our thesis, we thought the problem with Xeon Phi is it has a NUMA architecture. As our measurement shown, there is a NUMA factor between individual cores, but it is quite small. The slowest access from the most distant core is around 25% slower than the fastest access from the nearest core. Even though the NUMA factor is not so big, it might be beneficial to set the Xeon Phi as a NUMA system in Bobox’s scheduler.

As we shown in tests, the memory controllers are grouped by two. It means, there are four groups of memory controllers. We propose to create NUMA nodes around those four memory controller groups as shown in Figure 17. The NUMA nodes are in red ovals. Our enhancement should not be worse than existing solution, because we would preferably allocate memory from the controller that has the slowest latency from the cores placed in the node. Only time it could be slower is if we overload one memory controller and the others would be doing nothing.
However, in our opinion, a bigger problem is with caches and their synchronization through DTD. The biggest difference compared to standard NUMA or SMP architectures is that non-uniform cache architecture (NUCA) should not be ignored on big many-core architectures. Xeon Phi has to deal with this problem only if cache line is shared among cores.

The latency of cache accesses may vary in 1:3 ratio, depending on the data size. The smaller the data is, the more important is, which cores access it. For example, if we would have a shared structure (e.g. lock) for all cores or few of them, each core would have different access latency to this variable. This might lead to different times of processing of the same workload with different cores.
4.2.1. Bobox improvements

Bobox scheduler can be tuned by some parameters. They enable to set the number of thread groups and the number of threads per group. Before our thesis, there were some tests of Bobox with different parameters on Xeon Phi. The best results was achieved when using 4 thread groups. This corresponds to our proposal of having 4 NUMA nodes.

Those Bobox parameters set thread affinity and bound memory allocator to thread group. For better performance on Xeon Phi, we need to regulate, what physical addresses should be used for which core. That should set a NUMA for the coprocessor. If we wanted to move performance even further, we could use information about DTDs. For small shared structure, it would be beneficial to create special allocator which would allocate these data, according to the cores which would share it.
5. Conclusion

At first, we made a summary of all information that we were able to find about Xeon Phi’s internal architecture. We were interested in things related to core and memory, because these are crucial for every scheduler. We wrote benchmarks and based on their results we proposed adjustments to Bobox’s scheduler and allocator.

We wanted to find the cause of performance problem in Bobox. By benchmarking Xeon Phi, we found important architecture details that might result in bad Bobox’s scheduler behavior.

First of all, we have created picture of core’s position on the connection ring. We showed their position compared to memory controllers. We confirmed memory mapping between physical addresses and memory controllers by our benchmark. These all things were important in showing that there is a small NUMA factor on Xeon Phi. The difference between the fastest and slowest request is around 25%. Thanks to knowing the core and memory controllers layout, we were able to propose possible settings of NUMA nodes. Thanks to these information, Bobox’s scheduler can be adjusted to be able run faster. We have proposed, how to do it.

We have shown that there is difference around 1:3 in latency while accessing cache lines from different cores. Unfortunately, we were not able to find the cache line to DTD mapping. We showed that the non-uniform cache architecture should be taken into account, if we are sharing small data between cores and want the best performance on Xeon Phi. Generally, this is the biggest difference between standard NUMA/SMP architecture and many-core architecture. The effect of the non-uniform cache architecture cannot be ignored for in performance sensitive applications.

5.1. Future work

First thing is to modify the Bobox’s memory allocator to be able to allocate on specific physical addresses. Together with scheduler improvements, this allows to set NUMA on Xeon Phi properly.

There are still missing details about Xeon Phi architecture. One of them is a mapping between cache lines and DTDs. If we knew it, we would be able to create a NUCA-aware allocator and scheduler. This would allow us to benefit from faster cache latencies while sharing data between different cores.
Attachment

- **mcm** – directory with results of latencies between memory controllers and cores
- **mcm.c** – source code of mcm measurement
- **tdl** – directory with results of latencies between cache lines and cores
- **tdl.c** – source code tdl measurement
- **how_to_build_kmod.txt** – instructions, how to build a kernel module for Xeon Phi
- **Makefile** – makefile for kernel modules
Bibliography

List of Abbreviations

- VPU – Vector Processor Unit
- CRI – Core Ring Interface
- 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
- MESI – Modified Exclusive Shared Invalid protocol
- MOESI – Modifier Owned Exclusive Shared Invalid protocol
- PCI – Peripheral Component Interconnect
- I/O – input/output
- SMP – Symmetric Multiprocessing
- 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
• NUMA – Non-uniform Memory Access
• NUCA – Non-uniform Cache Architecture
• ID – Identification
• SSH – Secure shell
• SCP – Secure copy
• GOLS3