Communication in MyThOS

Whitepaper

Randolf Rotta, rottaran@b-tu.de
Vladimir Nikolov, vladimir.nikolov@uni-ulm.de
Lutz Schubert, lutz.schubert@uni-ulm.de

19th April 2016

Contents

1 Overview

Basically, three different scenarios can be identified, which require special communication directives: a) cross-core communication between processes running within the same CPU b) cross-cpu communication within a closely coupled multi-processor system (same host) and c) cross cpu-communication within a loosely coupled system constituted of processors distributed over several interconnected hosts.

The realization of the first required mechanism for cross-core communication in MyThOS is described in the global architecture whitepaper. For dynamic interactions between processes residing on different computational units (or hardware threads) *active messages* containing small tasks are used and transmitted between the HWTs. For that, three different priority-queues for kernel-, user- and idle-tasks are implemented and provided for each hardware thread. The communication is then realized by placing a new task into the appropriate queue of the destination thread. However, the operating system has to assure, that local and remote tasks have a consistent view on their computational environment, e.g. address space and data. This requirement is application-specific, since the application has to define which parts and components belong to the same protection level and address space. However, the OS controls the access to shared resources and ensures consistency.

For the second scenario there are several configuration possibilities. However, we assume a simplified model, where multiple processors (e.g. multiple Xeon Phi extension cards) reside in parallel within one and the same host computer. Parallel Xeon Phi cards, for example, are interconnected over PCIe and can share common physical address spaces. A special protocol (sCIF) then allows message based communication between processes running on the different Xeon Phi nodes over PCIe. MyThOS uses these mechanisms in order to provide a custom channel-based communication mechanism. Therewith, the communication between different MyThOS instances or components residing on different nodes and the host system is established in a common and controllable way. Such a scenario is discussed in more detail in chapter 2 where an implementation of a communication channel between the host system and a MyThOS instance running on a Xeon Phi card is explained.

The third scenario involves network based communication between OS instances. For the current state of the art MyThOS focuses on a single host machine. Network

based communication requires mythos-compatible drivers for network cards. The effort for the implementation of such communication mechanisms has been postponed, until the proposed architectural principles of MyThOS have been completely realized and verified for the single-node case.

2 Short Messages over PCIe2

The communication between host services, such as the application launcher and the MyThOS services on the many-core accelerators, is carried out via asynchronous message passing over PCIe2. As already explained, a similar mechanism can be established between different MyThOS instances or components hosted on different processors, which are backed by shared memory. In this particular case, the majority of messages relates to control and configuration akin to remote method invocations or remote system calls. These require just small messages with a size limit in the order of a few cache lines.

Data transfers may need larger messages. One way to implement them, is splitting the transfer into multiple small messages, which does not need additional services. In the long term, dedicated mechanisms for large data transfers, for example via DMA engines, can be added. These would be coordinated through short messages.

Separate message channels are needed for each direction, that is from host to accelerator and from accelerator to host. On the producer site, support for multiple concurrent producers is desirable. This ensures, that all threads can send messages without the need for locking based mutual access or forwarding to a proxy thread. On the consumer side, such support for concurrent access is not strictly necessary. A main thread would be responsible to pull new messages and convert them into local tasks. A single consumer queue can be combined with a non-blocking mutual exclusion mechanism to hand over the work to an idle thread.

The PCIe 2.0 network allows to map the accelerator's physical memory into the logical address space of applications running on the host processor. The host service can directly write and read the accelerator's memory almost like normal shared memory. However, careful protocol design is necessary because the PCIe 2.0 network does not keep the caches coherent, atomic operations like fetch-and-add are not supported, and the involved processors may have a different view on the logical and physical address spaces.

The interaction across multiple address spaces implies the need for memory pinning. The host operating system and MyThOS must be instructed to never migrate the frames that contain the communication channel data. Otherwise, processors will read from and write to wrong positions, which are not observed by their counterparts.

The easiest way to achieve this is by placing all shared communication data on the MyThOS side. There we, know the physical address of the channel and can ensure that the data will not be moved. For shared data on the host side, which runs a Linux system for example, two steps are needed: First, the allocated pages have to be pinned in order to prevent migration and, second, their physical addresses have to be provided to the MyThOS side. These need to be translated then into accelerator-physical addresses in order to become accessible from MyThOS.

2.1 Logical Structure: Slotted Circular Buffer

Using a fixed size memory area is the easiest approach for host to accelerator communication because just this area needs to be mapped into the logical address spaces once. Hence, a slotted circular buffer design was chosen: The messages are stored in a fixed size array with slots of fixed size (a few cache lines). The producer and consumer sides use independent counters for their logical read and write positions. The actual positions are modulo the number of buffer slots. Each slot contains a state header for flow control between producers and consumers. The producer writes sequentially into different slots of the circular buffer while the consumer tries to catch-up the producer counter. The flow control information is provided for each slot and it is used to establish an access protocol between the producer and the consumer, which will be explained later in this chapter.

The unidirectional channel is split into three data structures: ProducerSide, ConsumerSide, and ChannelData. This allows to place the respective parts into the appropriate memory, which enables the local use of atomic operations. Just the ChannelData is in the shared memory, which restricts it to ordinary read/write access and C++11 atomic load and store operations.

The ProducerSide and ConsumerSide objects contain the respective logical read/write positions and a pointer to the the ChannelData according to the local logical address space. No access to the other end's state is needed.

2.2 Deployment

For host to accelerator direction, the ProducerSide is placed in host memory while ConsumerSide and ChannelData are placed in the accelerator's memory. For the reverse direction, the ProducerSide and ChannelData are placed in the accelerator's memory and the ConsumerSide is placed in the host memory.

The host has the ChannelData mapped with CacheDisabled and WriteCombining flags. The write combining makes sending more efficient. Ideally, the incoming channel's data should be mapped with caching enabled. Unfortunately, the currently used driver interface does not support this mode.

2.3 Dynamic Interactions

Initially, the producer and consumer side logical positions are zero. The slot headers are initialized accordingly to be recognized as "free".

A producer acquires a logical write position by atomically incrementing a local write counter. This supports concurrent producers. A producer then polls the header of a respective slot until the slot is marked "free" (by being equal to the acquired logical position). Then, the message data is written into the slot and, finally, the slot is marked as "occupied" by setting a respective bit.

The consumer polls the slot of its current logical read position. If the slot is marked as "occupied", the data is copied or processed directly. Then, the consumer writes the slot's next logical position into the header in order to mark it as free and selects the next responsible producer at the same time. Finally, the consumer has to invalidate the slot from its L1 and L2 caches and increment the logical read position.

Access to shared memory over PCIe is not cache coherent. Reads to remote memory can be cached in the local cache, but writes to this memory will not invalidate these replica. Hence, the memory has to be mapped in `Cache Disabled` mode or the communication protocol has to implement proper manual cache line evictions. Usually, writes over PCIe should invalidate the destination's caches automatically. Unfortunately, currently this does not seem to work for writes from host to XeonPhi. Hence, the `CLEVICT0` and `CLEVICT1` instructions were used for manual cache line evictions.

The `CLFLUSH` and `MFENCE` instructions are not available on the XeonPhi. The new `CLEVICTx` instructions evict the requested cache line from the local L1 or respectively L2 cache but do not broadcast the evict request to other caches. Hence, all threads that read from `ChannelData` have to evict their data before reading the next message.

Future: Concurrent consumer support is possible by using a second bit in the slot header to mark the slot as locked by a consumer. This is done with an atomic compare-and-swap such that just a single consumer can succeed. All other consumers proceed to the next slots. Unfortunately, compare-and-swap is available only on the side that has the `ChannelData` in its local memory.

2.4 Implementation

Because the cache invalidation works differently on host and XeonPhi, two different implementations of the of the producer and consumer sides of the circular buffer are needed. These versions can be also be adapted more easily to specific needs of the respective the side. In our current implementation `ProducerSide` is specialized in the classes `PCIeRingProducerX86` (for the host) and `PCIeRingProducerPhi` (for the MyThOS side). Both extend a generic interface `ISendChannel`. Likewise, a `ConsumerSide` is implemented in the classes `PCIeRingConsumerX86` and `PCIeRingConsumerPhi`. Both extend the interface `IRecvChannel`.

The basic interface provided by `ISendChannel` has the methods `acquireSend()` and `trySend()`. The acquire method returns the logical write position, which is then handed to the send method. `trySend()` has to be repeated for the provided position until it succeeds. This non-blocking interface allows higher software levels to intermix polling for incoming messages.

The basic interface provided by `IRecvChannel` contains the methods `hasMessages()`, `tryRead()`, and `finishRead()`. The first method can be used to check whether there is a pending message without trying to read or write anything. `tryRead()` returns a pointer to message data and might lock an appropriate slot for concurrent consumer support. After a message has been processed, its address is provided to `finishRead()`, which flushes the respective caches and marks the slot as free.

Implementation Note: The `ChannelData` structure is generic and has a slot type attribute as well as a slot count attribute. The slot type is combined with the channels slot header to define the actual slots and slot size. In consequence, the ProducerSide and ConsumerSide classes get the actual ChannelData type as template type argument.

3 Current Specializations

The described mechanisms for message based communication between host and MyThOS over PCIe2 were employed in order to provide a) debug and status information from MyThOS to the host side and vice versa b) to enable remote configuration and data transfer from the host to a MyThOS instance. Both mechanisms resulted in small communication frameworks, which will be summarized in the following on a conceptual level, without diving into their implementational details.

3.1 Textual Debug and Binary Trace Output

MyThOS components produce a configurable amount of textual debug and log messages via the `mlog` subsystem. The `mlog` subsystem is integrated within all mythos modules and provides user interfaces for creating, filtering and sending log messages to customizable log sinks. One such sink is for example the debug output channel to the host. Currently the system is configured to send all levels of debug information of all subsystems over the PCIe2 channel to the host, where it can be further processed, e.g. stored into a file or printed on a terminal console. A per-module configuration of a debug-level is realizable as well.

Textual messages have a variable unbounded length and should be able to contain multi-line messages. Besides this, all hardware threads can produce messages concurrently and their output should be sequentialized for better readability (a multiple producers channel). Writing messages to a single output channel can be a blocking operation, because the communication is just one-way and, thus, no deadlocks are possible. Performance is not important because production systems would disable almost all debugging outputs.

In order to reduce the implementation effort and potential for errors, the short message channel from Section 2 is reused. Just a more specific message format had to be defined.

Several different `mlog` channels for different subsystems, e.g. memory management and userspace, are provided. Additional channels can easily be added. Each channel can output messages of different priority levels, as indicated by the programmer,

which can be filtered before output. Thereby, developers can specify to see only important messages from the system, while getting detailed information from the subsystem, they are currently working on, without altering existing code. Textual debug messages are stored into an output buffer together with a timestamp, the originating hardware thread ID, the number of remaining messages and the length of the message. This output buffer can then be accessed externally to output the messages. Trace messages use the same `mlog` infrastructure, but are distinguished from debug messages by a type identifier. A different output buffer is used for trace messages to improve performance.

An appropriate terminal application on the host implements the receiver side of the debug communication channel. The process needs to run with superuser access rights, in order to be capable of reading and writing to the channel regions within the mapped physical address space of the extension card. Every received message is processed according to its type and printed on screen. The output can be of course also forwarded to a log file.

3.2 Remote Method Invocation and System Calls

A special remote method invocation interface was also realized based on the PCIe2 communication mechanisms in order to allow for configuration and application deployment within a running MyThOS instance. Basically, a set of MyThOS system calls is exposed to the host over special stubs and proxies:

1. `allocPhysicalArea` - allocate an area of physical memory on the MyThOS side
2. `writePhysicalArea` - write into a previously allocated memory area on the remote side
3. `zeroPhysicalArea` - overwrite a given memory region with zeros
4. `mmap` - map a phsyical memory region into logical address space
5. `mprotect` - set up protection properties for a memory region
6. `exec` - create a user space thread within MyThOS and start execution from a given logical address

These system calls were used for the implementation of a MyThOS application launcher residing on the host side. The application code and initial data are read

from according ELF binaries at the host system and written (`writePhysicalArea`) segment-wise over the PCIe2 channel to dynamically allocated remote memory areas (`allocPhysicalArea`) within MyThOS. Those areas are then mapped into a logical address space for the application (`mmap`). The application execution can then be triggered with the *exec* remote system call, which accepts a virtual address for the application entry point (e.g. its *main* method).

A special server mechanism has been implemented, which runs as a kernel process on the MyThOS side and listens on the channel for remote method invocations. Therefore, an appropriate message type is provided, which basically contains an identifier of the invoked remote method and its parameters. Special server- and client-side stubs were implemented for each method, taking care of appropriate parameter serialization and deserialization. For the transfer of bigger data amounts a fragmentation mechanism automatically splits the data into chunks, which are then streamed over the channel. Based on continuation mechanisms results of the method invocations can be returned back to the caller on the host side. The first implementation of this remote system call interface supports only synchronous method invocations, however, an asynchronous variant based on future objects can be realized as well.

Bibliography