Förderkennzeichen:	01IH130003
Vorhabensbezeichnung:	MyThOS Modulares Betriebssystem für Massiv Parallele Anwendungen

MyThOS D5.2 Infrastruktur und Evaluierungsergebnisse

Stefan Bonfert, Vladimir Nikolov, Robert Kuban, Randolf Rotta

28. März 2017

Zusammenfassung

Dieses Deliverable beschreibt die Infrastruktur für die diversen Tests und dokumentiert die erzielten Evaluierungsergebnisse.

Das diesem Bericht zugrunde liegende Vorhaben wurde mit Mitteln des Bundesministerium für Bildung und Forschung unter dem Förderkennzeichen 01IH13003 gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt bei den Autoren.

Inhaltsverzeichnis

1 Introduction

In this document, we describe the infrastructure used for the evaluation of MyThOS along with the results of its evaluation. As an operating system tailored to both HPC applications and computing cloud scenarios, MyThOS mainly aims at fast thread instantiation times along with fast and flexible system call execution for dynamic and flexible resource sharing among applications. Therefore, we evaluated the latency for the creation of new ECs (software threads) as well as the latency of different system call variants for interaction with the operating system and the customization of its behavior.

2 Infrastructure

For the evaluation of MyThOS we utilized a server equipped with an Intel©Xeon Phi™5100 series coprocessor card. MyThOS is executed as an independent operating system on the coprocessor. The host computer is used to trigger the execution of a MyThOS application and to collect debug outputs containing evaluation results.

The Xeon Phi card is considered as a well suited evaluation platform due to its large number of independent control flows. It features 60 cores with 4 hardware threads per core. Therefore it can be used to evaluate the scalability of individual algorithms, entire applications or operating systems.

3 Evaluation Results

For the creation of new threads we capture three different values: The creation time, the deployment time and the time until execution. The creation time includes the allocation of memory buffers, different memory mappings, setup of communication structures and the setup of the thread itself. Therefore, it includes the overhead of five system calls and their execution. The deployment time describes the latency to communicate with the designated hardware thread, deploying the thread and report the operation's success back to the original one. The time until code execution is the time from the call of the thread creation function until the execution of the first line of user code in the newly created thread. The results of the creation of 5000 threads spread over all available hardware threads are shown in Figure 1.

On average, the creation of a new thread in MyThOS including all supporting structures takes 5.79×10^4 CPU cycles. All involved operations can be executed on the local core. In contrast, for deployment of the thread to a destination hardware thread, remote communication is required. The deployment time also includes the time for the transmission of the kernel-internal tasklet to the destination and back to the initiator to return the result of the operation. Therefore, the deployment is significantly more costly and takes 1.07×10^5 cycles. On the destination hardware thread, code can be executed right after the tasklet arrived there and before its transmission back to the sender. This allows the duration until the first code execution to be shorter than the deployment latency with 7.01×10^4 cycles. For comparison, we measured the duration from the creation of a new thread until the first code execution in Linux using pthreads. Similar to the MyThOS case, this includes the setup of all required data structures as well as the deployment of the thread. On average, this operation takes 3.62×10^5 cycles in Linux on the same hardware platform. This gives MyThOS a performance gain of factor 5.

To evaluate the system call performance, we benchmarked two system call variants. For the first one we instantiate a kernel object on a fixed hardware thread. All calls to this kernel object are pinned to this hardware thread and therefore are only executed there. For the benchmarks, we used a dummy kernel object, that immediately returns inside the system call handler. Thereby, we are able to measure the pure overhead of a system call. We measured the latency for a system call to this object, including the transmission of the reply, over different distances. The results are shown in Figure 2

A call to a kernel object located on the same hardware thread as the caller EC requires no communication and takes 2.57×10^3 cycles. In the case of remote system

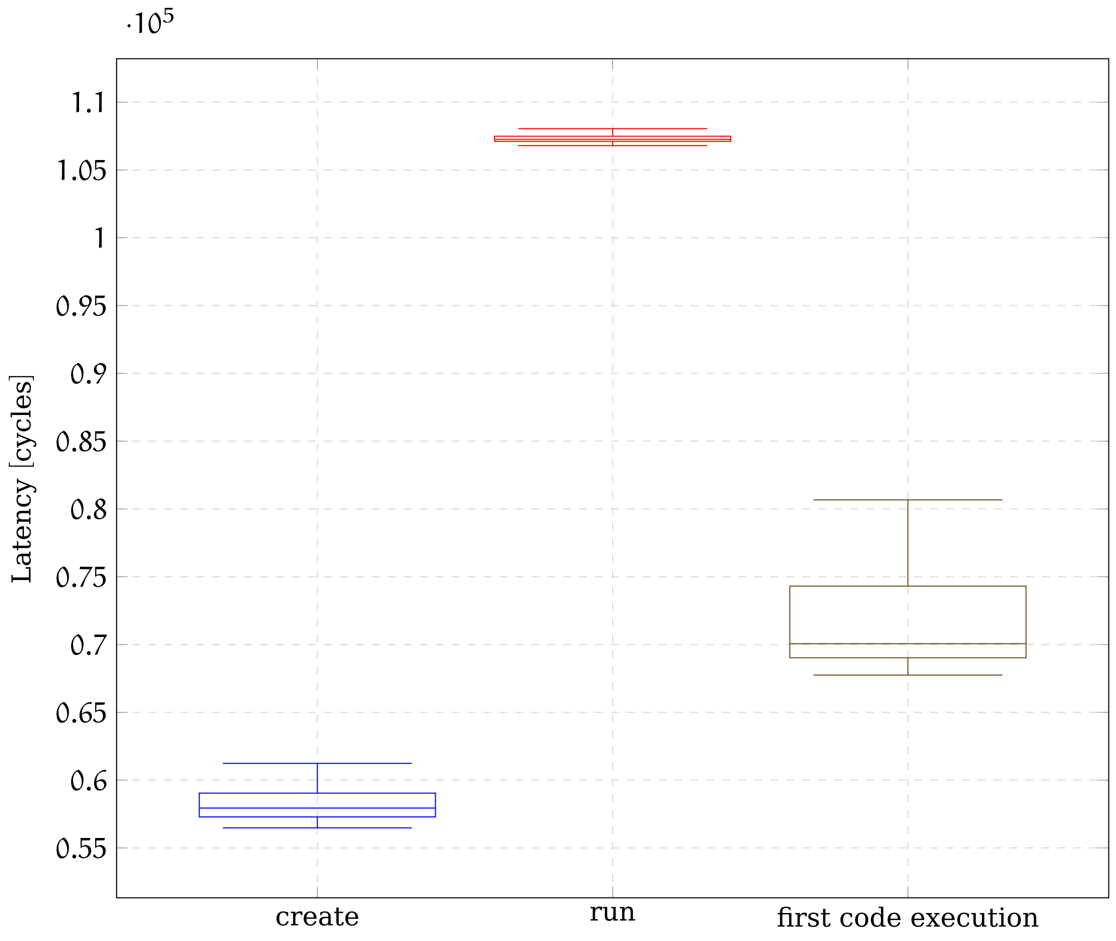

Figure 1: Latency to create and run an execution context

call execution, additionally endpoint resolution and communication costs play a major role, which is why the response times are much higher in these cases. For system call execution on a different hardware thread on the same core we measure a latency of 1.02×10^5 cycles, which is only a little lower than in the case of execution on a different core with 1.06×10^5 cycles. The ability for the user to specify the location of execution for a kernel object is a major difference between MyThOS and Linux, since Linux does neither support such fine-grained control over the execution of system calls nor the remote execution of them. Therefore, no fair comparison to Linux can

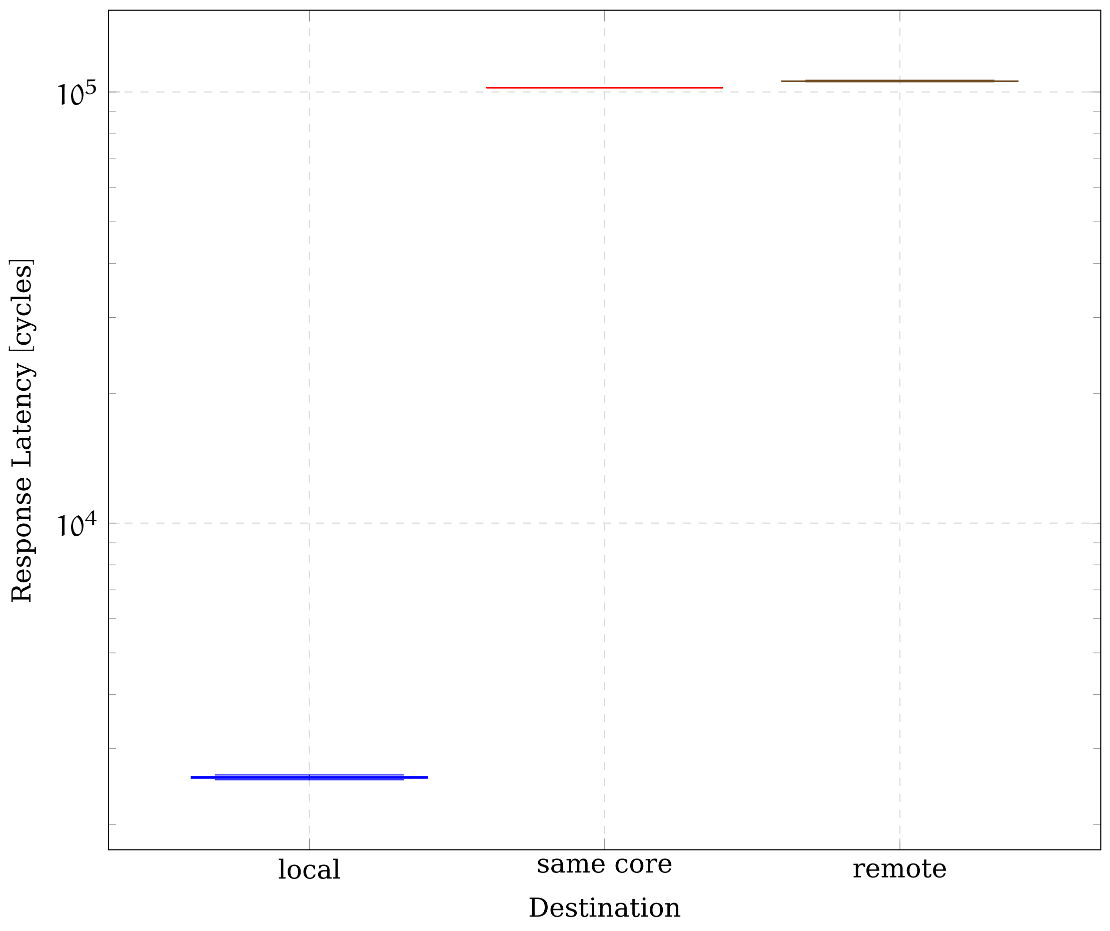

Figure 2: Response Latency for a kernel object with fixed location over different distances

be made in this respect.

The second system call variant we benchmarked includes mobile kernel obejcts. These objects are executed wherever they are first called to increase the spatial locality from caller to callee. If they are already in execution at a different location, the request is enqueued at this location in order to reduce the number of cache misses and thereby increase temporal locality. This execution semantic leads to different behavior, when looking at different workloads within the system call handler. We therefore benchmarked the call latency with both no workload and a loop counting from 1 to 1.00×10^5. The latter represents a system call carrying out some work within the kernel. We used this kernel object to issue system calls with a varying number of concurrently active ECs, that are iteratively using the kernel object. This setup was chosen to evaluate the scalability properties of MyThOS' system calls. The results are shown in Figures 3 and 4.

Figure 3 shows the latency until the system call initially returns to the caller. At this point, the system call was not necessarily executed, but it has been registered inside the kernel and delegated to the responsible hardware thread for execution. In the case of low workload in the system call, its execution is finished quickly and the object's execution can be moved to the caller. Therefore, for low workload no remote communication is necessary and the call returns to the caller in the order of 1×10^4 CPU cycles with no dramatic increase if more ECs are concurrently accessing the object. In the case of high workload in the kernel object, the kernel object tends to stay at its current point of execution across multiple calls, because one call is not done processing when the next call arrives at the object. Thereby, temporal locality is increased. Therefore, in this case remote communication between different hardware threads is required, leading to higher initial return times in the order of 1×10^6, during which the caller is blocked. However, there is almost no increase of this latency with a growing number of concurrent accesses to the kernel object. This indicates good scalability of the system calls implemented in MyThOS, as a single EC is not blocked longer, if more other ECs are accessing a operating system service at the same time.

Figure 4 shows the latency until a reply from the kernel object is received, i.e., the duration until the system call was processed and the result was transmitted back. These numbers include the return latency that was described above and therefore represent the complete processing time of a system call from the initial call of the function in userspace, until the reply is available to the application. In the case of low workload we experience the same behavior as for the initial return. The call's results are quickly available, because there is no contention at the kernel object. The system exhibits good scalability properties with only a slight increase in the measured latency, while the numbers stay in the area of 1×10^4. In the case of high workload, a linear increase in reply latency was observed. In this case, the kernel object is temporarily blocked by a previous call, whenever a call arrives. Therefore, an arriving call first has to wait until all previous calls have been processed. Since the length of this queue is proportional to the number of concurrent accesses, the

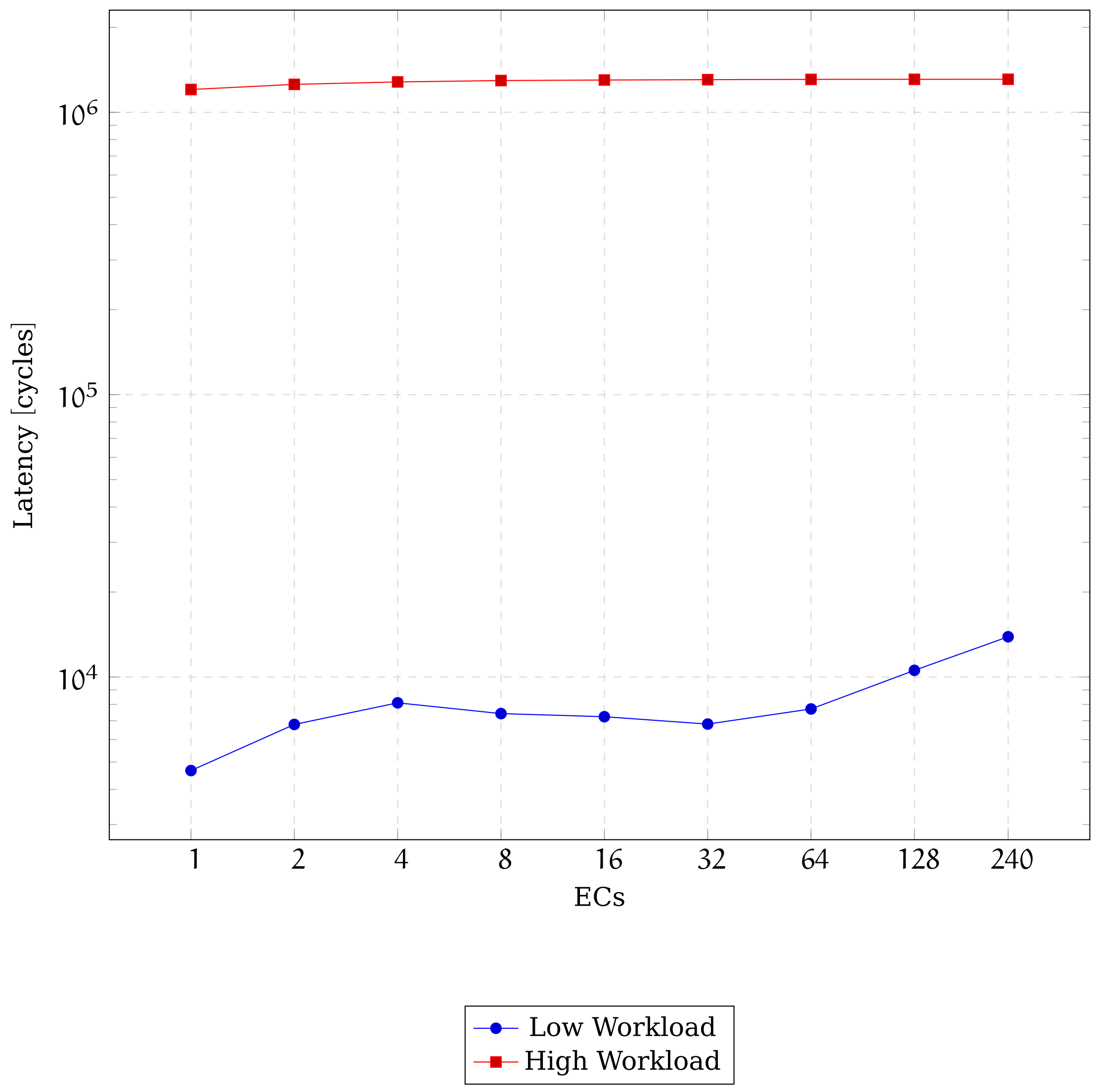

Figure 3: Latency of system call to a kernel object

observed latencies behave in the described linear fashion. Again, this indicates good scalability of the system call architecture, since no other bottlenecks are limiting the execution. The serialized execution can not be avoided by the operating system and a linear increase therefore is the best possible result. However, the user can create

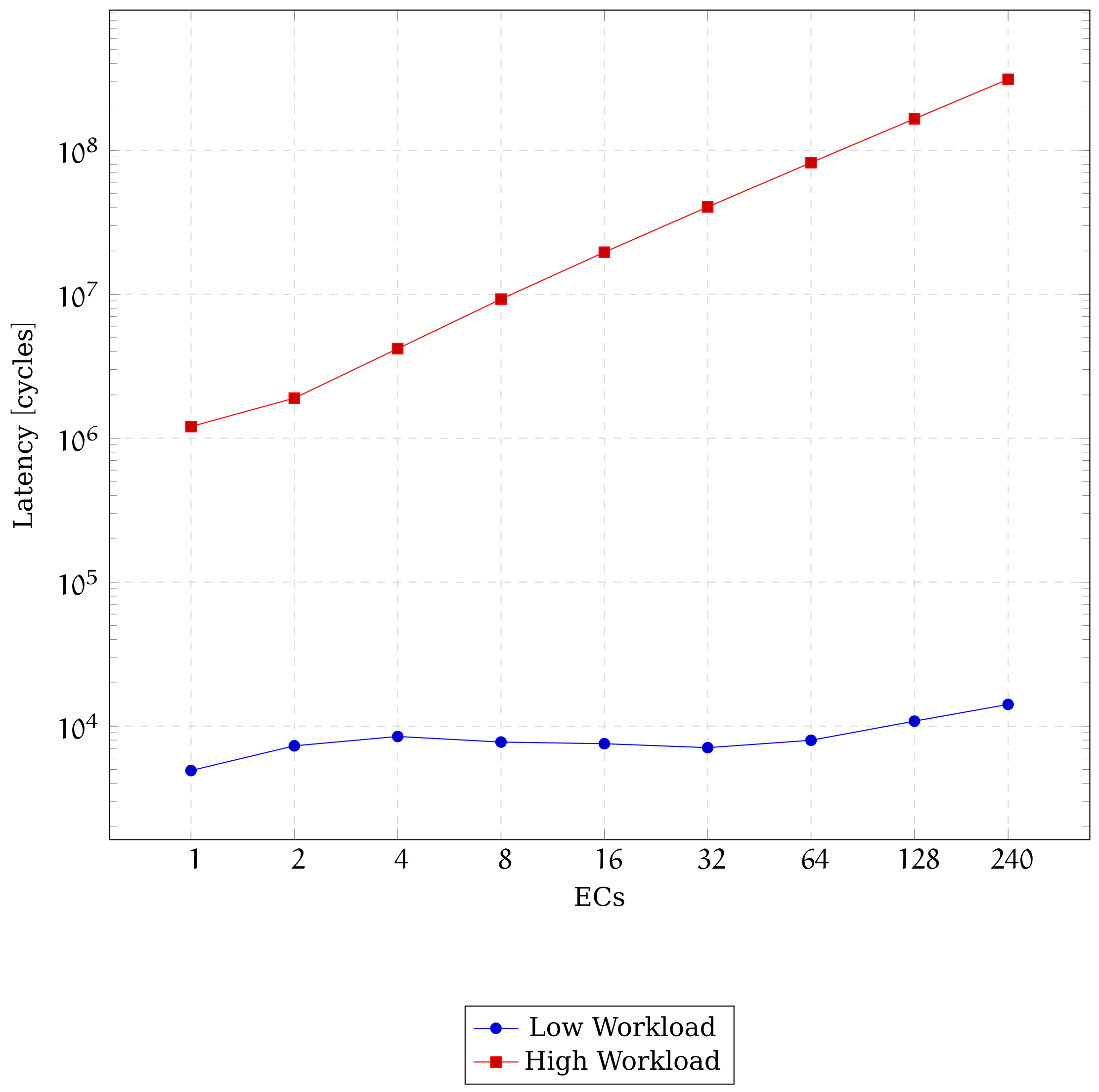

Figure 4: Duration until a reply is received

multiple instances of a kernel object and thereby reduce the time until the result is available. The approach of MyThOS is not to include this decision into the operating system, but leave it up to the user to configure the operating system to fit its needs as good as possible.