Chapter 1

System Software Research Team

1.1 Members

Yutaka Ishikawa (Team Leader)
Masamichi Takagi (Senior Scientist)
Atsushi Hori (Research Scientist)
Balazs Geroﬁ (Research Scientist)
Yoshiyuki Morie (Research Scientist)
Masayuki Hatanaka (Research & Development Scientist)
Takahiro Ogura (Research & Development Scientist)
Tatiana Martsinkevich (Postdoctoral Researcher)

The system software research has been conducted in cooperation with the following members of the System Software Development Team in the Flagship 2020 project.

1.2 Research Activities

The system software team focuses on the research and development of an advanced system software stack not only for the "K" computer but also for toward exa-scale computing. We have been mainly focusing on scalable high performance communication and file I/O libraries and/or middlewares. The former research topics, sharing virtual address space and IHK/McKernel light-weight kernel, have been almost taken over by the System Software Development Team, but the research results are shown here.

1.3 Research Results and Achievements

1.3.1 Persistent Collectives

Collective communication latency is a major concern, as it has become longer with the increase of scale of parallel computers. Overlapping computation and communication is one effective method for hiding communication delay. Recent networks have offloading functionality, so non-blocking communication is more important. Although non-blocking collective communication exists as an overlapping method, it requires generating a command sequence every time a non-collective communication is issued. In contrast, persistent non-blocking collective communication can be separated into the initialization and the collective communication body, so the initialization can be removed from the loop. In such a persistent non-blocking collective communication, the sequence of communication commands is generated at the time of initialization. The communication command sequence is then reused for each collective communication instead of the sequence needing to be re-generated. Moreover, if the sequence can be offloaded to a network device, then more efficient execution is possible without
CHAPTER 1. SYSTEM SOFTWARE RESEARCH TEAM

using CPU cycles. Although some implementations of persistent non-collective communication exist, there is no implementation of persistent non-collective communication that uses the offload functionality.

The offloading functionality was added to the Tofu2 interconnect on the FX100 supercomputer, a successor to the K computer. In this paper, an offloaded persistent non-collective broadcast for the Tofu2 interconnect is implemented. We will report the performance improvement from offloading persistent non-blocking collective communication on a real machine.

Tofu2 has a framework for hardware-driven distributed execution of communication commands called session mode CQ (SMCQ). As shown in Figure 1.1, SMCQ consists of a transmit order queue (TOQ) and three pointers: the producer, consumer, and scheduling pointers. The TOQ descriptor has the information for RDMA communication such as command type, source, destination, and session progress step (SPS) field. The producer pointer points to the end of command sequences enqueued by the application. The consumer pointer indicates the end of commands executed. When an RDMA engine processes an incoming RDMA operation packet, the scheduling pointer can be advanced by the SPS fields of the packet. Therefore, the progress of the command processing in SMCQ is driven by incoming packets and does not need the help of the CPU.

1.3.1.1 Offloaded Persistent Broadcast

We implemented the offloaded persistent broadcast for FX100[6,7]. This implementation targets the Trinaryx3 algorithm used in the K computer. This is an algorithm for improving the communication bandwidth by mapping multiple pipeline paths for the torus network. The Trinaryx3 algorithm is an extension of the Trinary algorithm in which broadcast data are divided into three fragments, each of which is transmitted using the Trinary algorithm. The three pipeline paths of the trinary tree are mapped to the 3D torus in Figure 1.2. Each pipeline communication is offloaded and executed by SMCQ.

Figure 1.3 shows the performance of an offloaded persistent non-blocking broadcast on the FX100. The offloaded persistent non-blocking broadcast is better than the non-blocking broadcast because the persistent non-blocking collective is able to complete creating the sequence of communication commands and exchanging the remote memory information at MP_Bcast_init(). Figure 1.4 shows the execution times of MPI_Bcast_init and MPI_Start. MPI_Bcast_init is relatively expensive and should not be repeated. Also, the cost of MPI_Start is sufficiently small, indicating that offloading is effective.
1.3. RESEARCH RESULTS AND ACHIEVEMENTS

1.3.1.2 Offloaded Persistent Neighborhood alltoall

This implementation targets 2D Cartesian Halo exchange pattern[3]. It uses four SMCQs (one SMCQ per MPI_Sendrecv exchange) in the pattern.

Figure 1.5 shows the communication latency on FX100 for four types of persistent neighbor alltoall collective implementation. In this figure, the horizontal axis shows the message size in the halo exchange and the vertical axis shows the elapsed time from MPI_Start to MPI_Wait. The NAAP, the offload version, is 60% better than P2PP at 8 bytes message size.

<table>
<thead>
<tr>
<th>Name</th>
<th>init</th>
<th>start</th>
<th>test</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAAP</td>
<td>Tofu2 API</td>
<td>Tofu2 API</td>
<td>Tofu2 API</td>
</tr>
<tr>
<td>P2PP</td>
<td>MPI_SendRecv,init</td>
<td>MPI_Startall</td>
<td>MPI_Testall</td>
</tr>
<tr>
<td>P2PN</td>
<td>n/a</td>
<td>MPI_SendRecv</td>
<td>MPI_Testall</td>
</tr>
<tr>
<td>NAAN</td>
<td>n/a</td>
<td>MPI Neighbor_alltoall</td>
<td>MPI_Test</td>
</tr>
</tbody>
</table>

Figure 1.5: Latency Benchmark of Persistent Neighborhood All-to-All

1.3.2 Data Transfer Framework

This section describes the progress in the work on the Data Transfer Framework. The framework targets multi-component applications in which components are loosely coupled via files written using PnetCDF file I/O library. By using the DTF such components can silently replace file I/O with direct sending of data between the components and thus, improve the performance of the application.
1.3.2.1 Introduction

Multi-component workflows, where one component performs a particular transformation with the data and passes it on to the next component, is a common way of performing complex computations. Using components as building blocks we can apply sophisticated data processing algorithms to large volumes of data. Because the components may be developed independently, they often use file I/O and the Parallel File System to pass data. However, as the data volume increases, file I/O quickly becomes the bottleneck in such workflows. In this work, we propose an I/O arbitration framework called DTF to alleviate this problem by silently replacing file I/O with direct data transfer between the components. DTF treats file I/O calls as I/O requests and performs I/O request matching to perform data movement. Currently, the framework works with PnetCDF-based multi-component workflows. It requires minimal modifications to applications and allows the user to easily control I/O flow via the framework’s configuration file.

1.3.2.2 Overview

The DTF aims to provide users of multi-component workflows with a tool that would allow them to quickly switch from file I/O to direct data transfer without needing to cardinally change the source code of the components.

First, the user must provide a simple configuration file that describes the file dependency in the workflow (example in Figure 1.6). It only needs to list the files that create a direct dependency between two components, i.e. if the components are coupled through this file. The DTF intercepts PnetCDF calls in the program and, if the file for which the call was made is listed in the configuration file as subject to the data transfer, the DTF handles the call accordingly. Otherwise, PnetCDF call is executed normally.

In order to transfer the data from one component to another, we treat every PnetCDF read or write call as an I/O request. The data transfer is performed via what we call the I/O request matching. First, designated processes, called I/O request matchers, collect all read and write requests for a given file. Then, each matcher finds out who holds the requested piece of data by matching each read request against one or several write

Figure 1.6: DTF configuration file

```
[INFO]
ncmp=2  ! number of components
comp_name="rdr"  ! component name
comp_name="wrt"
ioreq_distrib_mode="range"  !divide by dim length
buffer_data=0

[FILE]
filename="restart.nc"
writer="wrt"  !component that writes to the file
reader="rdr"  !component that reads from the file
iomode="memory"  !enable direct transfer
```

Figure 1.7: I/O request matching. Request matchers are marked with a red shape outline. For simplicity, only one reader process is showed to have read I/O requests.
1.3. RESEARCH RESULTS AND ACHIEVEMENTS

requests. Finally, the matcher instructs the processes who have the data to send it to the corresponding process who requested it. All the inter-process communication happens using MPI. We note that here we differentiate between the PnetCDF non-blocking I/O requests and the DTF I/O requests, and we always assume the latter unless stated otherwise.

The I/O patterns of the component that writes to the file and the component that reads from it may be drastically different, however, dynamic I/O request matching makes DTF flexible and allows it to handle any kind of I/O patterns transparently for the user.

The three main API functions provided by the DTF are the following:

- `dtf_init(config_file, component_name)` - initializes the DTF. The user must specify the path to the DTF configuration file and state the name of the current component which should match one of the component names in the configuration file;
- `dtf_finalize()` - finalizes the DTF;
- `dtf_transfer(filename)` - invokes the data transfer for file `filename`;

A simplified example of a writer and reader components is presented Figures 1.8 and 1.9, as well as their common DTF configuration file (Figure 1.6). To enable the direct data transfer it was enough to add three lines of code to each component — to initialize, finalize the library and to invoke the data transfer — and provide a simple configuration file.

1.3.2.3 Performance evaluation

Next, we show how the DTF performs with a real world workflow application — SCALE-LETKF. SCALE-LETKF is a real-time severe weather prediction application that combines weather simulation with assimilation of weather radar observations. It consists of two components — SCALE and LETKF — that are developed independently. SCALE is a numerical weather prediction application based on the ensemble simulation; LETKF performs data assimilation of real-world observation data together with simulation results produced by SCALE. In each iteration, SCALE writes the simulation result to the Parallel File System (PFS) using the Parallel NetCDF API. The files are subsequently read by LETKF. After LETKF finishes assimilating the observation data, the output is written to files which become the input for SCALE in the next iteration.

In the chosen test case LETKF assimilates the data from a Phased Array Weather Radar with a resolution of 500 m. The number of processes participating in one ensemble simulation is fixed to nine processes in all tests, the total number of processes per component is nine multiplied by the number of ensembles.

The size of the history and restart file in one ensemble is constant, we change the total amount of I/O by varying the number of ensembles from 25 to 100. Table 1.1 contains the information about the amount of data written and read in each configuration. In all tests, every ensemble process in SCALE writes 363 MB of data, out of which LETKF process requires only about one quarter. A SCALE process generates 255 write requests, LETKF process — 31 read request.

The results show that the DTF helps to improve the total execution time of SCALE-LETKF (Figure 1.10) by cutting on the I/O time. In the largest execution with 100 ensembles, the I/O time was improved by a factor of 3.7 for SCALE and 10 for LETKF.
Table 1.1: Cumulative I/O amount in SCALE-LETKF

<table>
<thead>
<tr>
<th>Number of ensembles</th>
<th>Total write size (GB)</th>
<th>Total read size (GB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>79.78</td>
<td>18.97</td>
</tr>
<tr>
<td>50</td>
<td>159.56</td>
<td>37.94</td>
</tr>
<tr>
<td>75</td>
<td>239.35</td>
<td>56.91</td>
</tr>
<tr>
<td>100</td>
<td>319.13</td>
<td>75.88</td>
</tr>
<tr>
<td></td>
<td>Per process 363 MB</td>
<td>86.3 MB</td>
</tr>
</tbody>
</table>

Figure 1.10: Runtime

Figure 1.11: SCALE I/O

Figure 1.12: LETKF I/O

For more details on the design of the DTF and more experimental results please refer to [5].

1.3.3 Sharing Virtual Address Space

The two most common parallel execution models for CPUs today are multiprocess and multithread. The multiprocess model allows each process to own a private address space, and the multithreaded model shares all address space by default. We propose a new implementation of the third model, called Process-in-Process (PiP), where multiple processes are mapped into a single virtual address space. Thus, each process still owns its private storage but can directly access the storage of other processes in the same virtual address space. The idea of address-space sharing is not new. What makes PiP unique, however, is that its design is completely in user space, making it a portable and practical approach for large supercomputing systems.

PiP allows multiple parallel tasks to execute the same or different programs in a shared virtual address space environment while maintaining privatized variable sets. The PiP task does not follow the definition of conventional process or thread. For instance, PiP tasks share the same virtual address space whereas processes have isolated virtual address space; two PiP tasks can execute arbitrary programs in parallel in the same virtual address space, whereas threads must be derived from the same program. Moreover, a process or PiP task always starts its execution from the main() function, but a thread starts its execution from an arbitrary function. PiP defines a special task called root process that owns the virtual address space and spawns multiple tasks executing in the same VAS as of the root, and the model is thereby named Process-in-Process.

Table 1.2: Experimental platform information

<table>
<thead>
<tr>
<th>Name</th>
<th>CPU</th>
<th># Cores</th>
<th>Clock</th>
<th>Memory</th>
<th>OS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wallaby</td>
<td>Xeon E5-2650 v2</td>
<td>8×2(×2)</td>
<td>2.6 GHz</td>
<td>64 GB</td>
<td>CentOS 7.3 (and McKernel)</td>
</tr>
<tr>
<td>OFP</td>
<td>Xeon Phi 7250</td>
<td>68×4</td>
<td>1.4 GHz</td>
<td>96(+16) GB</td>
<td>CentOS 7.2</td>
</tr>
<tr>
<td>Albireo</td>
<td>ARM Cortex A57</td>
<td>8</td>
<td>2.0 GHz</td>
<td>16 GB</td>
<td>CentOS 7.3</td>
</tr>
<tr>
<td>The K Computer</td>
<td>SPARC64 VIIIIfx</td>
<td>8</td>
<td>2.0 GHz</td>
<td>16 GB</td>
<td>XTCOS</td>
</tr>
</tbody>
</table>

Table 1.2 shows the hardware and software environments where PiP is being tested to run on. We design two execution modes in the PiP implementation;

**Process Mode:** In this mode, the OS kernel thread and its stack can be created by calling the Linux clone() system call. In this mode, PiP tasks behave like normal processes regarding PID, termination, file descriptors, and signal handling.

**Thread Mode:** In this mode, the pthread_create() POSIX function is used and PiP tasks behave like threads regarding TID, termination, file descriptors, and signal handling. Again, PiP provides the variable
privatization even in this thread execution mode.

Figure 1.13 compares the time to spawn null tasks by using PiP, Pthread, fork()&exec(), vfork()&exec(), and posix_spawn() on the machines listed in Table 1.2. As shown in this figure, the PiP spawning times are mostly the same as those with creating processes.

![Figure 1.13: Task spawning time on four platforms](image)

1.3.3.1 DOE-MEXT Collaboration

To prove the efficiency of PiP, MPICH is prototyped to utilize PiP. This work has been done in the context of DOE-MEXT research collaboration. The modifications in the prototyped PiP-aware MPICH are: 1) Hydra process manager to spawn MPI processes as PiP tasks, 2) enhanced PT2PT rendezvous protocol to have only one memory copy for intra-node communication, 3) single-copy MPI datatype communication, and 4) modified MPI_Win_alloc_shared to have shared memory regions.

![Figure 1.14: Intel MPI Benchmark PingPong Performance Improvement](image)

In [4], a variety of PiP micro-benchmark and macro-benchmark results are reported. Figure 1.14 shows one of them, the PingPong bandwidth differences between the original MPICH and PiP-aware MPICH. As shown in this figure, PiP succeeds to boost the bandwidth, especially on the Xeon processor. Among the evaluation results, it should be noted that PiP can be used for in-situ applications for having better communication performance between simulation program and in-situ program.
1.3.3.2 CEA-Riken Collaboration

It is known that new parallel execution models, where virtual address space is shared among tasks and each task has a privatized variable set, can implement efficient intra-node communications and save memory. MPC developed by CEA is very similar to PiP in terms of providing the new execution model.

To have deeper knowledge on MPC and PiP, the source codes are shared and investigated on both sides. MPC allocator, which is another malloc library developed in the MPC project, can be used in PiP in which an allocated memory region by a task cannot be freed by the other task. This problem is already solved in MPC. In this first project year of the collaboration between CEA and Riken, MPC and PiP focused on the evaluation of the tools on each side, and proposing collaborations axes on the MPC allocator. The investigation of embedding MPC allocator into PiP reveals that the current MPC allocator cannot be embedded into PiP without source code changes, which have been identified.

1.3.4 IHK/McKernel

This Section describes recent research results related to IHK/McKernel. Specifically, results of large scale evaluation will be provided in Section 1.3.4.1, published in [1]. An introduction to the PicoDriver fast-path device driver framework is given in Section 1.3.4.2, published in [2]. Finally, results of characterizing hardware performance variation across various hardware platforms using lightweight OS kernels are presented in Section 1.3.4.3, published in [8].

1.3.4.1 Large Scale Evaluation on Oakforest-PACS

We performed large scale evaluation of the IHK/McKernel lightweight multi-kernel operating system using up to 2,048 compute nodes of Oakforest-PACS. Without striving for completeness we provide some of the results we obtained.

![Figure 1.15a](image)

Figure 1.15a shows single node experiments as a baseline for further measurements. The figure shows relative performance obtained on McKernel compared to Linux. As seen, McKernel outperforms Linux on all benchmarks achieving up to 18% on the Lulesh benchmark.

The rest of the results presented here show select scaling measurements for the AMG2013, miniFE and CCS-QCD mini-applications, shown in Figure 1.15b, Figure 1.16a, and Figure 1.16b, respectively. As shown, McKernel outperforms Linux on these benchmarks by up to 21%, 3.5X and 38%, respectively. For further information on scaling results and their analysis, refer to [1].

1.3.4.2 PicoDriver: Fast-path Device Drivers for Multi-kernel Operating Systems

Standalone lightweight kernel operating systems in high-end supercomputing have a proven track record of excellent scalability. However, the lack of full Linux compatibility and limited availability of device drivers in LWKs have prohibited their wide-spread deployment. Multi-kernels, where an LWK is run side-by-side with Linux on many-core CPUs, have been proposed to address these shortcomings. In a multi-kernel system the LWK implements only performance critical kernel services and the rest of the OS functionality is offloaded to Linux. Access to device drivers is usually attained via offloading. Although high-performance interconnects are commonly driven from user-space, there are networks (e.g., Intel’s OmniPath or Cray’s Gemini) that require
1.3. RESEARCH RESULTS AND ACHIEVEMENTS

We have recently proposed PicoDriver [2], a novel device driver architecture for multi-kernels, where only a small part of the driver (i.e., the performance critical piece) is ported to the LWK and access to the rest remains transparent via Linux. Our solution requires no modifications to the original Linux driver, yet it enables optimization opportunities in the lightweight kernel. We demonstrated that on 16,384 Intel Knight’s Landing (KNL) CPU cores (interconnected by OmniPath network) we can outperform Linux by up to 30% on various mini-applications.

For brevity, we include only two application results in this document, shown in Figure 1.18. The figure shows normalized performance to Linux on up to 256 compute nodes of Oakforest-PACS. As seen, McKernel with the HF11 PicoDriver ourperforms Linux by up to 30% on these applications.
1.3.4.3 Hardware Performance Variation: A Comparative Study using Lightweight Kernels

Imbalance among components of large scale parallel simulations can adversely affect overall application performance. Software induced imbalance has been extensively studied in the past, however, there is a growing interest in characterizing and understanding another source of variability, the one induced by the hardware itself. This is particularly interesting with the growing diversity of hardware platforms deployed in high-performance computing (HPC) and the increasing complexity of computer architectures in general. Nevertheless, characterizing hardware performance variability is challenging as one needs to ensure a tightly controlled software environment.

Recently, we have proposed to use lightweight operating system kernels to provide a high-precision characterization of various aspects of hardware performance variability [8]. Towards this end, we have developed an extensible benchmarking framework and characterized multiple compute platforms (e.g., Intel x86, Cavium ARM64, Fujitsu SPARC64, IBM Power) running on top of lightweight kernel operating systems. Our initial findings show up to six orders of magnitude difference in relative variation among CPU cores across different platforms.

We used a number of different platforms to characterize, Table 1.3 shows a summary of these architectures.

<table>
<thead>
<tr>
<th>Platform/Property</th>
<th>Intel Ivy Bridge</th>
<th>Intel KNL</th>
<th>Fujitsu FX100</th>
<th>Cavium ThunderX</th>
<th>IBM BG/Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISA</td>
<td>x86</td>
<td>x86</td>
<td>SPARC</td>
<td>ARM</td>
<td>PowerISA</td>
</tr>
<tr>
<td>Nr. of cores</td>
<td>8</td>
<td>64+4</td>
<td>32+2</td>
<td>48</td>
<td>16+2</td>
</tr>
<tr>
<td>Nr. of SMT threads</td>
<td>2</td>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
<td>4</td>
</tr>
<tr>
<td>Clock frequency</td>
<td>2.6 GHz</td>
<td>1.4 GHz</td>
<td>2.2 GHz</td>
<td>2.0 GHz</td>
<td>1.6 GHz</td>
</tr>
<tr>
<td>L1d size</td>
<td>32kB</td>
<td>32kB</td>
<td>64kB</td>
<td>32kB</td>
<td>16kB</td>
</tr>
<tr>
<td>L1i size</td>
<td>32kB</td>
<td>32kB</td>
<td>64kB</td>
<td>78kB</td>
<td>16kB</td>
</tr>
<tr>
<td>L2 size</td>
<td>256kB</td>
<td>1MB x 34</td>
<td>24MB</td>
<td>16MB</td>
<td>32MB</td>
</tr>
<tr>
<td>L3 size</td>
<td>20480kB</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>On-chip network</td>
<td>?</td>
<td>2D mesh</td>
<td>?</td>
<td>?</td>
<td>Cross-bar</td>
</tr>
<tr>
<td>Process technology</td>
<td>22nm</td>
<td>14nm</td>
<td>20nm</td>
<td>28nm</td>
<td>45nm</td>
</tr>
</tbody>
</table>

We found that the different architectures exhibited diverse behaviour for the SHA256 benchmark, shown in Figure 1.19. Despite the same L1 cache size and associativity, we observed no L1 data misses on the ThunderX platform but approximately 150k misses on the Intel Ivy Bridge platform. We decided to include the results as-is because we consider cache implementation details also micro-architecture-specific. Another reason is that the number of L1 misses on Ivy Bridge show little variation themselves. The wide base of the violins on FX100 and ThunderX already indicate that a lot of cores experience no variation at all, while Ivy Bridge performs...
significantly worse and KNL shows an order of magnitude more variation still.

![Figure 1.19: Hardware performance variation under the SHA256 benchmark.](image)

We expected the BlueGene/Q to be among the lowest variation platforms but our measurements do not reflect that. At this point we can only speculate that the 16 KiB L1 data cache and the only 4-way set associativity of the L1 instruction cache have influence on the performance variation. We reduced the cache fill level to 80\% so that auxiliary data such as stack variables have the same cache space in 32 KiB and 16 KiB caches, but we could not measure lower cache miss number of lower performance variation.

![Figure 1.20: Hardware performance variation under the DGEMM benchmark.](image)

Figure 1.20 shows results for the DGEMM benchmark, which measures floating point operations. This benchmark confirms the low variation of the FX100 and ThunderX platforms and the rather high variation of the Ivy Bridge, KNL and BlueGene/Q platforms. We saw high numbers of cache misses on the Ivy Bridge platforms and therefore reduced the cache pressure to 70\% fill level. We saw stable or even zero cache miss numbers for all cores of the Ivy Bridge platform, but variation did not improve.

### 1.3.5 Utility Thread Offloading Interface

The number of cores per node in HPC system has increased to hundreds. This abundance makes it possible to partition core resource and run different kinds of tasks in different partitions to increase application performance by parallelism and increase isolation between them by partitioning. Typical example of this technique is to run helper threads in a dedicated partition, such as MPI asynchronous progress threads and glibc asynchronous I/O threads. However, the resource management part of the technique is done manually. And thus it puts too much burden on user and the management is not portable across different systems or system settings. To respond to this situation, we introduce a standard abstract interface, called Utility Thread Offloading Interface (UTI). It focuses on the system where resource is divided into two partitions for two kinds of tasks, i.e., compute tasks and helper tasks, and focuses on the shared resource management with those two. It is designed as the first step toward the system-wide, shared resource management with multiple kinds of tasks and multiple partitions.

This is on-going work and the followings were done during this fiscal year.

1. Revision of the UTI API. This work was done in collaboration with Intel, Sandia National Lab and incorporating the feedbacks of Argonne National Lab. and CEA.
2. Prototype implementation of UTI kernel functions in IHK/McKernel.
3. Prototype implementation of UTI user-level library for IHK/McKernel.
4. Implementation of asynchronous progress thread using UTI in MPICH.

The first two topics are explained in the followings.

**UTI API**  The API consists of C functions and macros. The usage steps are described in the followings.

1. A runtime library (e.g. MPI library) creates a hint description object by calling `uti_attr_init()` and write the hints using the abstract expression, by using `UTI_ATTR_*` macros. The hints include preferred CPU locations and thread behaviors relating to the scheduling policy.

2. The runtime library passes the hints to kernel and lets it create a helper thread (or utility thread) by calling `uti_thread_create()`. The created thread is pthread-compatible.

3. Kernel finds the optimal CPU location and scheduling policy using the hints.

**IHK/McKernel implementation**  The implementation is divided into the following four functions.

- **Thread creation**
  1. A runtime library uses special system call to modify the behavior of `clone()` and then calls `clone()` to create a utility thread (call it mck-uti-thread)
  2. mck-uti-thread enters kernel mode and sends a context switch request with its user-context to Linux and waits for the system call request or exit request described below
  3. A `mcexec` worker thread takes the request (call this thread lin-uti-thread)
  4. lin-uti-thread creates a `ptrace` tracer process and loads the context. `ptrace` is used for the hook described below.
  5. The tracer process attaches to lin-uti-thread

- **System call offload to McKernel**
  1. The tracer process hooks a system call and sends a system call request to McKernel if it is one of the system calls which modify McKernel objects (e.g. `mmap`, `munmap`, `brk`, `futex`).
  2. mck-uti-thread waiting in kernel mode takes the request and perform the system call for lin-uti-thread

- **Signal**
  1. McKernel relays a signal to mck-uti-thread to Linux
  2. A `mcexec` worker thread sends the signal to lin-uti-thread

- **Exit**
  1. The tracer process detects the exit of lin-uti-thread and sends an exit request to McKernel
  2. mck-uti-thread waiting in kernel mode takes the request and calls the exit function

1.4 Schedule and Future Plan

We continue to enhance performance and functionality of our system software stack for K, FX100, Post-K, and other manycore architectures, such as Intel Xeon Phi.

1.5 Publications


