An Implementation of User-Level Processes using Address Space Sharing

Atsushi Hori
Center for Computational Science
RIKEN
Tokyo, JAPAN
ahori@riken.jp

Balazs Gerofi
Center for Computational Science
RIKEN
Tokyo, Japan
bgerofi@riken.jp

Yutaka Ishikawa
Center for Computational Science
RIKEN
Tokyo, JAPAN
yutaka.ishikawa@riken.jp

***Abstract*—There is a wide range of implementation approaches to multi-threading. User-level threads are efficient because threads can be scheduled by a user-defined scheduling policy that suits the needs of the specific application. However, user-level threads are unable to handle blocking system-calls efficiently. To the contrary, kernel-level threads incur large overhead during context switching. Kernel-level threads are scheduled by the scheduling policy provided by the OS kernel which is hard to customize to application needs. We propose a novel thread execution model, *bi-level thread*, that combines the best aspects of the two conventional thread implementations. A bi-level thread can be either a kernel-level thread or a user-level thread at runtime. Consequently, the context switching overhead of a bi-level thread is as low as that of user-level threads, but thread scheduling can be defined by user policies. Blocking system-calls, on the other hand, can be called as a kernel-level thread without blocking the execution of other user-level threads.**

Furthermore, the proposed bi-level thread is combined with an address space sharing technique which allows processes to share the same virtual address space. Processes sharing the same address space can be scheduled with the same technique as user-level threads, thus we call this implementation a *user-level process*. However, the main difference between threads and processes is that threads share most of the kernel state of the underlying process, such as process ID and file descriptors, whereas different processes do not. A user-level process must guarantee that the system-calls always access the appropriate kernel information that belongs to the particular process. We call this *system-call consistency*.

In this paper, we show that the proposed bi-level threads, implemented in an address space sharing library, can resolve the blocking system-call issue of user-level threads, while at the same time it retains system-call consistency for the user-level process. A prototype implementation, ULP-PiP, proves these concepts and the basic performance of the prototype is evaluated. Evaluation results using asynchronous I/O indicate that the overlap ratio of our implementation outperforms that in Linux.

***Index Terms*—Threads, Concurrent Programming, Concurrent programming structures**

I. Introduction

The debate between User-Level Threads (ULT) are Kernel-Level Threads (KLT) has taken place for decades. The characteristics of ULT and KLT mentioned below are part of most computer science textbooks.

The most well-known kernel thread implementation is PThreads. Since KLTs are created and scheduled by an OS kernel, when a KLT is blocked by calling a blocking system-call, another KLT is scheduled by the OS, if any. The context switching overhead between KLTs is high because the OS kernel is involved in this operation. ULT libraries, on the other hand, are implemented at user-level. A ULT can be created by allocating a new stack region and switching to it by using for example `setjump/longjump` in Unix, `ucontext` in Linux, or `fcontext` implemented in the Boost C++ library [1]. There is no kernel-level operation involved during context-switching between ULTs and thus the context switch is faster than that of KLT. Unlike KLT, a ULT is blocked when it calls a blocking system-call and no chance to schedule the other eligible-to-run ULTs.

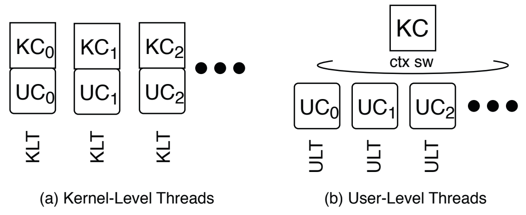


Fig. 1. KLT and ULT decomposed

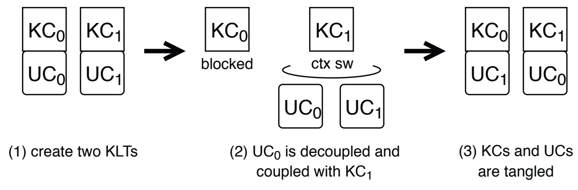

KC0
KC1
UC0
UC1
KC0
blocked
KC1
ctx sw
UC0
UC1
KC0
KC1
UC1
UC0
(1) create two KLTs
(2) UC0 is decoupled and coupled with KC1
(3) KCs and UCs are tangled

Fig. 2. Coupling and decoupling of UCs and KCs

Let us consider KLTs and ULTs in more detail. A KLT can be though of as a pair of kernel context (KC) and user context (UC) as shown in Figure 1. A KC is the reference for accessing resources maintained by an OS kernel. A UC is the same as a ULT. A KLT always consists of a pair of a KC and a UC and is often categorized as 1:1 thread execution model. What if we could decouple KC and UC of a KLT? The decoupled UC now becomes a ULT and can be scheduled by another KLT. One might wonder what happens with the decoupled

KC. The decoupled KC has nothing to do but idling (i.e., it either becomes blocked in the OS kernel or it can busy-wait) as shown in Figure 2.

If a UC is coupled with a KC and the KC has no other UC to schedule, then the UC and KC can be thought of as a KLT. If a UC is one of the UCs scheduled by a KC then the UC can be thought of as a ULT. By coupling a UC and a KC, the ULT becomes a KLT. By decoupling UC from a KC in a KLT, the KLT becomes a ULT. In this paper, we call this implementation a *Bi-Level Thread (BLT)* because a BLT can be either a KLT or a ULT at runtime. It should be noted that the proposed BLT technique can be applied to any ULT implementations, independent from ULP.

We implement the proposed BLT in the Process-in-Process (PiP) [2] library and name it *ULP-PiP*. PiP is a pure user-level library enabling for processes to share the same virtual address space. Unlike KLTs which also share the same virtual address space, all variables defined in the *process* on PiP are privatized. Here, *variable privatization* means that there are N instances of variable `x` when N processes are derived from the same program defining the `x`. KLTs share all variables, however, all variables in PiP are not shared but shareable. Any objects in PiP are accessible and shareable since everything is located in the same virtual address space. For more implementation details of PiP, refer to Section IV. The same technique for switching ULTs can be applied to switching process contexts in the same address space. We call this *User-Level Processes (ULP)*. The overhead of context-switching between processes is larger than that of PThreads, because each process has its own address space and the address spaces must be switched when to switch one process to the other. Thus, the low overhead of switching ULPs is more beneficial than that of ULTs.

Although the context switching technique to implement ULT and ULP can be the same, there is a big difference between them. A system-call is to access resources inside of an OS kernel, and each KC has a different set of resources. For example, when a UC calls the `getpid()` system-call, the returned PID may vary depending on the scheduling KLT. If the `open()` system-call is called, then the opened file descriptor (FD) is only valid if the KC calling `open()` and the KC calling `read()`, for example, are the same. This *system-call consistency* must be maintained by a ULP system.

In this paper, we make the following contributions:

1) We propose a novel thread execution model called Bi-Level Thread (BLT) where ULT can become KLT and KLT can become ULT,
2) BLT can resolve the blocking system-call issue found in ULT,
3) which we combine with an address space sharing technique and propose User-Level Processes (ULP),
4) we introduce the concept of system-call consistency for ULPs, and
5) through micro-benchmark results we demonstrate how BLTs and ULPs outperform conventional programming models.

II. Background

Using a conventional ULT implementation, the blocking system-call problem can be avoided by issuing asynchronous I/O (AIO) system-calls, such as `aio_read()` or `aio_write()`. The current Linux AIO implementation works as follows; 1) a PThread is created at the first call of `aio_read()` or `aio_write()`, 2) the main thread delegates the I/O operation to the created thread, and 3) it waits for the completion of the I/O by calling `aio_return()` or `aio_suspend()`. When a conventional ULT tries to read or write on a file, it calls the corresponding AIO call instead and waits for its completion in a loop consisting of yield and the call to `aio_return()`. Unfortunately, the current AIO infrastructure only supports read and write. There are no corresponding calls for `open()`, `listen()`, `connect()`, and numerous other blocking system-calls. The nonblocking I/O might be another solution to I/O operations for ULTs, however, it requires more programming effort.

Scheduler Activation (SA) was proposed to cope with the blocking system-call issue of ULTs [3]. SA, however, requires the cooperation of the OS kernel and the ULT system, and SA is not widely supported by today's OS kernels. In contrast to SA, our proposal is implemented at entirely at user-level.

In general, thread execution models can be categorized into three configurations; 1:1, N:1, and M:N, where the first number is the number of UCs and the second number is the number of KCs in the system. The execution model of the proposed BLT can be any of these by changing KLTs to ULTs or ULTs to KLTs at runtime (shown in Figure 3). The current prototype implementation of BLT only allows to create BLTs as KLTs at beginning. Thus the number of UCs and KCs must be the same. In this sense, our proposed BLT can be categorized as N:N model. This restriction comes from the system-call consistency, i.e., every BLT is able to preserve the consistency. However, the proposed BLT model can be easily extended to M:N model. This will be discussed in Section VII.

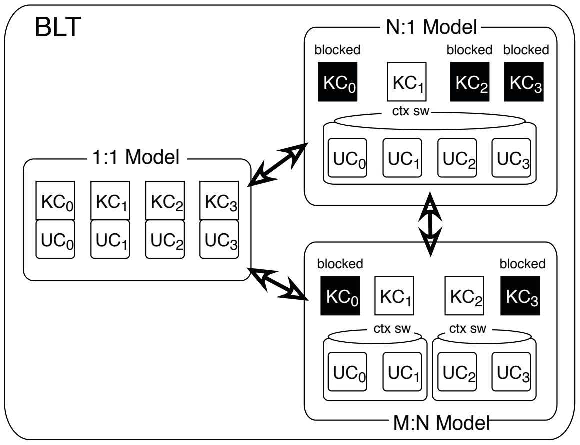


Fig. 3. BLT Execution Model

Most threading libraries, regardless of KLTs or ULTs, follow the fork-join model for creating threads and synchronize at thread termination. BLT, however, does not follow this model since BLTs must be created as KLTs from which some may become a ULT. This BLT behavior resembles a thread pool model better than the fork-join model. Additionally all BLTs always terminate as KLTs coupled with their original KC. Here, the *original* KC is defined as the KC which was used to create the KLT (i.e., a pair of KC and UC) in the beginning. Thus in the parent process that creates its child processes as BLTs in ULP, the `wait()` system-call can be used to wait for BLT terminations, just like the way used to wait for `fork()`ed processes.

To summarize the relation of BLTs and ULPs:

1) A BLT is created as a KLT consisting of a pair of UC and KC
2) The KC created at the beginning is called original KC
3) UC and KC of a BLT can be decoupled and the decoupled UC becomes a ULT
4) When a UC and a KC are coupled, the UC becomes a KLT
5) If a KC has no UCs eligible to run, then the KC becomes idle (by calling a blocking system-call or busy-waiting)
6) If an idle KC is given a UC to be scheduled, then the KC is unblocked and starts running the given UC
7) When a UC terminates, it is coupled with its original KC to become a KLT and the KLT terminates
8) A BLT implementation can support ULPs when combined with an address space sharing technique (e.g., PiP)

III. Related Work

Address space sharing technique, where processes share the same virtual address space and maintaining privatized variables has been being proposed; SMARTMAP [4], MPC [5] , Shinjuku [6], PVAS [7], and PiP [2]. SMARTMAP, Shinjuku and PVAS require designated OS kernels to support the address space sharing. MPC is more like a language system consisting of compiler and linker. MPC is based on PThread but its execution entity look like a process. The MPC compiler converts user defined static variables to thread local variables for each (fake) process to have privatized variables. Shinjuku supports preemptive scheduling for ULTs. A signal is sent to KLTs when to preempt. Among the shared address pace sharing implementations listed above, PiP is the only one implementation implemented as a pure user-level library. So PiP is the most portable and practical among the other implementations. Kaiming et al. proposes a new framework of work stealing in MPI by using the address space sharing (PiP) [8].

ULP [9] is implemented on top of PVAS. This *ULP* switches TLS regions when switching process contexts. Although the address space sharing technique has the potential to implement ULP, however, none of them addresses the system-call consistency issue.

TMPI [10] proposed the two-context switching technique which is almost the same with the UC/KC coupling and decoupling described in this paper. The two context switching takes place when a ULT calls an MPI function inside of which a blocking system-call is called. Although their intention, not to block ULTs while calling blocking system-calls, is the same with us, however, they did not try to generalize their technique to BLT and ULP.

There are many ULT implementations; Qthreads [11], MassiveThreads [12], Argobots [13], to name a few. Their concern is mostly scheduling and work stealing. Castelló et al., thoroughly summarized ULT implementations [14]. Argobots implemented their own TLS system while the others do not support TLS.

Scheduler Activation, K42 [15], and PM^2 [16] can handle the blocking system-call issue with the cooperation between OS kernel and user-level thread library. Each of them consists of user-level thread library and kernel API. SA was implemented on NetBSD [17] and some other OSes, unfortunately this SA support by NetBSD is obsolete now.

OpenMP introduced nested parallel loops and tasking which may create threads dynamically and often the number of threads exceeds the number of CPU cores. This oversubscribed situation may cause the large overhead due to the number of thread context switching. If ULT is used for underlying OpenMP runtime, instead of using PThreads, then this overhead can be reduced. BOLT [18] is an OpenMP runtime system using ULT to tackle this problem.

MPI is a widely-used communication library in the HPC community. The gap between computation speed and the communication latency is getting bigger and bigger every year, the latency hiding technique becomes more important. Current MPI supports asynchronous communication though, it forces users to have more programming effort and there are cases where users are unable to hide latencies completely. This situation may become more severe for larger-scale parallel computing. The larger the network, the harder the prediction of communication behavior. Another approach for this latency hiding is over-subscription. Similar to the OpenMP case described above, context switching overhead can be problematic when using oversubscribed KLTs or processes. Thus, MPI implementations using ULT are gathering attentions. MPIQ [19] and AMPI [20] are the ones for such purpose.

As described above, over-subscription is gathering attentions in the OpenMP and MPI implementations. In the MPI+X hybrid parallel programmings model, the 'X' has been considered as OpenMP, many researchers are working to replace and/or re-implement OpenMP with using ULT. It is the authors' wish that this paper will contribute to such work. The reason why we focus on ULP is motivated by MPI [21]. Although the MPI standard does not define how *MPI processes* [1] are implemented, i.e., whether it is multi-process or multi-thread, most MPI implementations are based on multi-process execution model and most MPI applications assume this model. Therefore, ULP is a more suitable execution model than ULT. Another possible application of ULPs are in-situ programs and multi-physics simulations. In a typical in-situ case, the in-situ program is attached to a simulation

[1]Each MPI process is an execution entity to have an MPI rank.

program to run simultaneously. Theoretically it is possible to merge the in-situ program and the simulation to have a single program so that it run on a ULT system, but this approach is impractical. Merging different programs can come at significant effort, especially if those programs are written in different programming languages. It would be more convenient to run them as separate programs.

IV. Process-in-Process (PiP)

In this section, we briefly introduce PiP which is the authors previous work, because this is one of the key technologies to implement our ULP-PiP. The motivation of PiP development is to have a new innode parallel execution model expected to be more suitable for many-core architectures as oppose to the conventional multi-thread (e.g., OpenMP) and multi-process (e.g., MPI) execution models. In the current multi-thread model, threads can easily communicate with the others, however, there is overhead to protect simultaneous updates on a variable because all variables are shared among threads. Contrastingly in the multi-process model, each process has its own variable set, however, efficient inter-process communication is hard to implement because a process cannot access data owned by the other processes. SMARTMAP, MPC, Shinjuku, PVAS and PiP enables processes to share the same virtual address space and to privatize variables on each process.

This programming model looks similar to the shared memory model but it does not. Shared memory is to share a physical memory region. Each process has to map the shared physical memory and each process may have different logical address of the shared memory region, resulting the pointers to the shared memory region cannot be dereferenced without offset. In contrast, the address space sharing is to share the entire address space. Every process has exactly the same logical to physical address mapping, in other words, a whole page table representing the address space is shared from the beginning. There is no need of creating a new memory map to share, and pointers can be dereferenced as they are. Furthermore, the minor page faults to create page table entries happen only once per page in the address space sharing regardless to the number of involving processes, whereas the minor page faults happen to every process accessing shared memory pages in the shared memory model.

PiP is implemented purely at the user-level and this makes PiP portable and practical. All the other address space sharing techniques require specialized OS kernel (SMARTMAP, MPC, Shinjuku, and PVAS) or language system consisting of compilers and some other related tools (MPC).

PiP utilizes the `dlmopen()`, not a typo of `dlopen()`, glibc function to load and dynamically link PIE (Position Independent Executable) program and required shared object files. Unlike `dlopen()`, the `dlmopen()` function can create a new name space to link variables and functions between shared object files. Thus all variables can be privatized by creating a new name space.

The program loaded by calling `dlmopen()` can run as a process by calling Linux's `clone()` system-call, or can run as a PThread by calling the `pthread_create()` glibc function. Former is called *process mode* and latter is called *thread mode* in PiP. In the process mode, the processes sharing the same address space are more like a normal process from the viewpoint of the OS kernel, each process may have its own signal attributes (handler mask, etc.) and parent process can wait the termination of spawned processes by calling the `wait()` system-call. In the thread mode, the PiP processes are basically PThreads in terms of the signal attributes and termination handling. Note that the variable privatization is effective in both PiP modes. Since the `clone()` system-call is very Linux specific, the thread mode is supported to run PiP on the environment not supporting the `clone()` system-call. The PiP library also provides several functions to hide the difference between the two PiP execution modes. For more details, refer to our paper [2][2]. All evaluations in this paper uses the process mode.

PiP root process is a normal Unix/Linux process and it can spawn PiP processes[3] in the same address space with that of the root process. The PiP processes must be derived from PIE programs. In an MPI implementation by using PiP, for example, the MPI process manager is the PiP root process and the MPI processes are the PiP processes spawned by the PiP root.

In theory, a program utilizing the heap segment cannot run with PiP. Since only one heap segment is allowed to have in one address space, the heap segment must be shared among PiP processes. Unfortunately the `(s)brk()` system-call is not designed to be shared (i.e., multi-thread safe). This heap segment issue is avoided by setting the `malloc` option not to use heap, instead to use `mmap` in the PiP library. In our PiP experiments, including running large simulation programs using MPI, we have not faced any problems to run existing programs with PiP.

V. Detailed Design of BLT and ULP

A. Trampoline Context

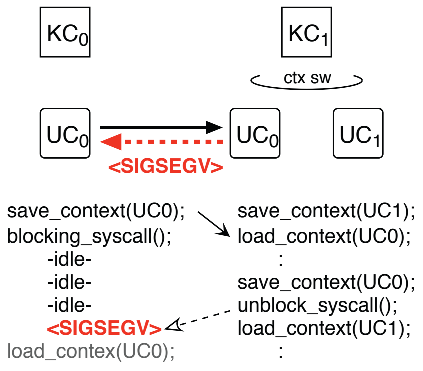
It is not difficult to implement decoupling UC from a KLT, saving the current UC and the saved context can be resumed by the some other KC. The KC that loose UC to run has nothing to do but idle. There are two problems here; a) how to idle the KC, and b) how the decoupled UC is coupled again with the idling KC. Let us examine one by one.

When a KC is busy-waiting while idling, it must run on a stack. The UC which was associated with the KC is already saved to run with the other KC. The UC cannot be scheduled to run because the original KC is busy-waiting on the same stack. Well, this busy stack problem would be solved by blocking the KC by calling a blocking system-call. In this case, however, another problem arises when the UC is back to the original KC. If the UC has been scheduled by the other KC, then

[2]PiP is an open source software package and able to download from https://github.com/RIKEN-SysSoft/PiP

[3]In paper [2], they are called *PiP tasks* since they do not follow the process nor the thread definition. In this paper, the term *PiP process* is used just for simplicity.

the stack state (stack pointer and stack content) is changed. The context of the original KC includes the stack pointer assuming the stack state of the UC is not changed. Thus it is also impossible to schedule the UC by the original KC, when the UC has been scheduled to run by the other KC.



```
save_context(UC0);       save_context(UC1);
blocking_syscall();      load_context(UC0);
      -idle-                    :
      -idle-             save_context(UC0);
      -idle-             unblock_syscall();
<SIGSEGV>                load_context(UC1);
load_contex(UC0);               :
```

Fig. 4. Decoupling Issue

Figure 4 shows the example of this problem. Decoupling can be implemented as; 1) save UC_0, 2) pass the saved UC_0 to KC_1 so that the KC_1 can schedule it as a ULT, and 3) KC_0 calls a blocking system-call to suspend. Once the decoupled UC_0 is scheduled by KC_1 and run, the stack state of UC_0 has been changed and KC_0 cannot resume UC_0 execution.

The root cause of this problem is the context for a KC to idle. To avoid this problem, we introduce *trampoline context (TC)*. A trampoline context is another context used for the transition from KLT to ULT or vice versa. The stack region of a trampoline context can be very small. Every BLT has an associated TC. A trampoline context may be created at the time of a KLT creation, or in a lazy way when to decouple. Figure 5 shows how decoupling and coupling using TC work. In this figure, the procedure (1) to (3) are for the transition from KLT to ULT, and the procedure (4) to (6) are for the transition from ULT to KLT. When a KLT (KC_0 and UC_0) becomes ULT, it first switches to the associated trampoline context and the UC_0 is now free from KC_0 and can be scheduled by any other KC. Then TC_0 becomes idle and the transition completes. When a ULT (UC_0) becomes a KLT again (with KC_0), then KC_1 which is the current scheduler of UC_0 cuts off UC_0 so that it can be scheduled by the other KC, and wake KC_0. KC_0 is resumed with TC_0 and swap TC_0 and UC_0 contexts. Finally UC_0 becomes KLT running with KC_0. Note that the stack state associated with a TC is not changed by the other KC, and thus the above problem can be avoided.

B. System-call consistency

To deal with the blocking system-call issue of ULT and to maintain the system-call consistency, we introduced two library functions; `couple()` and `decouple()`. The `couple()` function is to couple the calling UC with its original KC, assuming the current UC is running as a ULT, i.e., scheduled by the other KC. The `decouple()` function

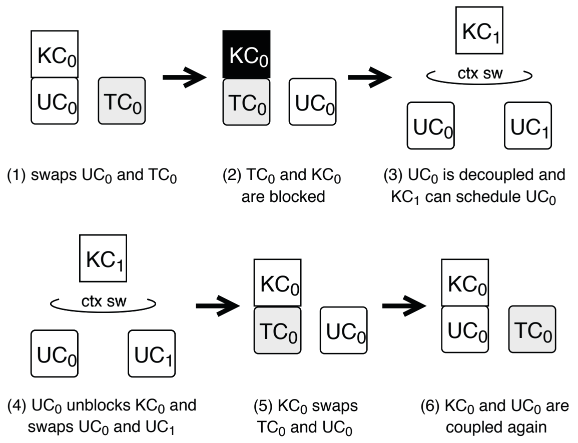

KC0
UC0
TC0
(1) swaps UC0 and TC0
KC0
TC0
UC0
(2) TC0 and KC0 are blocked
KC1
ctx sw
UC0
UC1
(3) UC0 is decoupled and KC1 can schedule UC0
KC1
ctx sw
UC0
UC1
(4) UC0 unblocks KC0 and swaps UC0 and UC1
KC0
TC0
UC0
(5) KC0 swaps TC0 and UC0
KC0
UC0
TC0
(6) KC0 and UC0 are coupled again

Fig. 5. Trampoline Context

is the opposite, assuming the calling UC is running as a KLT. It decouples the KC and decoupled UC is enqueued so that the UC is going to be scheduled by the other KC. If a UC wants to call a blocking system-call or a series of system-calls, some of them are blocking, then the system-call or system-calls are enclosed by the `couple()` and `decouple()` functions. This is all that a user has to do. Then the code enclosed by these functions is executed by the same original KC and the system-call consistency is preserved.

Figure 6 shows an example of BLT usage scenarios, but not limited to. Here, CPU cores are divided into two groups; one is for running user program and another is dedicated to execute system-calls. BLTs are created to run user program and to act as a scheduler. The number of BLTs to run user program is larger than (over-subscription) or equal to (no-over-subscription) the number of CPU cores to run the program. BLTs to run user program are decoupled and decoupled UCs are scheduled by the scheduling BLTs. The number of scheduling BLTs are equal to the number of CPU cores for user program execution and each KC of the BLTs is bound to one of these CPU cores. The decoupled KCs from the BLTs to run program are bound to the CPU cores for system-call execution. Here, a CPU core for executing system-calls may have more than one KCs.

$$NC = NC_{prog} + NC_{syscall} \tag{1}$$

$$NB = NC_{prog} \times (O + 1) \tag{2}$$

Here, NC is the number of CPU cores, NC_{prog} is the number of CPU cores to run user program, $NC_{syscall}$ is the number of CPU cores dedicated for system-call execution, NB is the number of BLTs, and O is the over-subscription magnification. The number of CPU cores for system-calls must be chosen based on the frequency and duration of the system-calls.

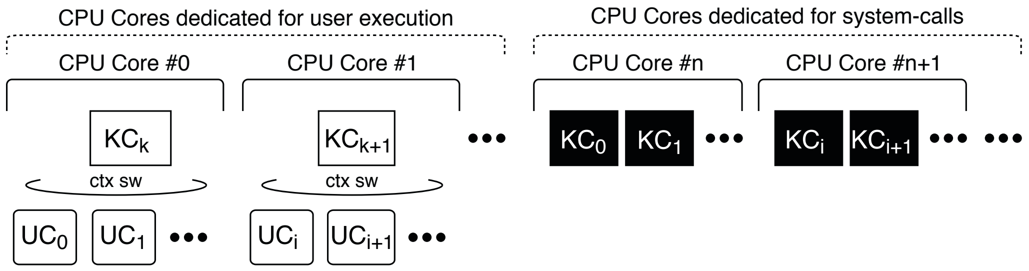


Fig. 6. A BLT Usage Scenario

In addition to the system-call consistency, thread local storage (TLS) regions[4] must also be switched when switching a UC to another because each process has its own TLS variables. Usually a TLS region is located right before or after the thread descriptor [22]. A designated hardware register is used to point the thread descriptor. On an x86_64 CPU, the `FS` segment register is used to point the thread descriptor and TLS variables are accessed by using this register. Unfortunately the `FS` register is privileged and the `arch_prctl()` system-call must be called to update the value of this register. Thus, the context switch overhead is increased considerably and this is the reason why most ULT implementations ignore TLS variables whereas ULP cannot.

Table I shows the more detailed procedure when to call a system-call and it is enclosed by the `couple()` and `decouple()`. This procedure starts from the state (3) shown in Figure 5, where UC_0 is running as a ULT scheduled by KC_1 and KC_0 is blocked already. The `swap_ctx()` function is to save registers of the current UC and load registers of the new UC. Here, UC_0 is about to call a system-call. The `couple()` function is to couple the current UC_0 with its original KC_0. If readers take a look at this procedure carefully, readers would notice that there are two race conditions. The first one is between Seq.3 of KC_1 and Seq.4 of KC_0. Here, the UC_0 context saved by KC_1 is loaded by KC_0. The second one is between Seq.8 of KC_0 and Seq.9 of KC_1. Here, the UC_0 context saved by KC_0 is loaded by KC_1. So we must have synchronizations on those two points.

To implement ULP, processes must share the same virtual address space. Unlike multi-thread, each process might be derived from different programs which might be written in different programming languages. Further, variables must be privatized so that each process has its own variable set. We chose Process-in-Process (PiP) to implement our ULP since PiP is purely implemented at user-level.

As already mentioned, TLS register which points to a TLS region must be switched when to switch contexts between different UCs. On an x86_64 architecture we have to call a system-call to do this, but this is totally depending on CPU architecture and how CPU registers are used by language systems. On an AArch64 (ARM 64bit) architecture, the TLS register (`tpidr_el0`) which can be accessed from user applications is used and TLS switching is much faster than that of x86_64. So the TLS switching overhead can be negligible on such architectures. Fortunately, the TLS register is set when a process or thread is created and once it is set it will not be altered again during the lifespan of a process or thread. In our implementation, TLS register content is saved at the time of creation of a ULP and new TLS register content is set at every context switching excepting the context switch between TC and UC.

[4]A TLS region holds TLS variables. A TLS variable can be defined by adding `thread_local` keyword. The most well-known TLS variable is `errno`.

VI. Evaluation

A. Evaluation Environment

The proposed BLT and ULP are implemented in the PiP library as to prove those concept. Thus, there is a room for further optimization and the API is tentative at the time of this writing. Since our system is very portable, evaluations in this section was carried out by using two machines shown in Table II, one is x86_64 named *Wallaby* and another is AArch64 named *Albireo*. The context switching is implemented by using the `fcontext` in the Boost C++ library (Version 1.72.0)

Some evaluations in this section, clock cycles are also measured on x86_64 using the `RDTSC` instruction. Unfortunately AArch64 does not have the instruction to obtain the clock cycles and no clock cycles is measured on Albireo.

All evaluations in this section has a warming up loop followed by a measurement loop. All values are the minimum ones of ten runs, because the evaluations in this section have no theoretical fluctuation but OS noise. Indeed, the fluctuations in all evaluations were quite small.

B. Basic Performance

Table III shows the basic performance numbers on each machines largely depending on CPU architectures, measuring the time to user-level context switch and the time to load a value to the TLS register, the `FS` register on Wallaby (x86_64) by calling the `arch_prctl()` system-call, and the `tpidr_el0` register on Albireo (AArch64).

The `fcontext` context sizes are 64 bytes on x86_64 and 88 bytes on AArch64, and it takes only few 10s of nanoseconds to swap contexts (save context to the stack and load context from the stack) on both machines. The times to load the TLS register have a large difference. This is because

TABLE I
DETAILED PROCEDURE OF `couple` AND `decouple`

Seq.#	User Code (UC_0)	KC_1 Library Code	KC_1 User Context	KC_0 Library Code	KC_0 User Context
0	:	[running]	UC0	[being blocked]	TC_0
1	couple()	enqueue(UC_0,KC_0)	:	:	:
2		unblock(KC_0)	UC_0	[unblocked]	:
3		swap_ctx(UC_0,UC_i)	$UC_0 \rightarrow UC_i$	UC_0 = dequeue()	TC_0
4		[running]	UC_i	swap_ctx(TC_0,UC_0)	$TC_0 \rightarrow UC_0$
5	system_call()	:	:	system_call()	UC_0
6	decouple()	:	:	enqueue(UC_0,KC_1)	UC_0
7		[yield or suspend]	:	swap_ctx(UC_0,TC_0)	$UC_0 \rightarrow TC_0$
8		UC_0 = dequeue()	UC_i	[blocking itself]	TC_0
9		swap_ctx(UC_i,UC_0)	$UC_i \rightarrow UC_0$	[being blocked]	:
10	:	[running]	UC_0	:	:

TABLE II
EVALUATION ENVIRONMENT

Name	Wallaby	Albireo
Architecture	x86_64	AArch64
CPU Type	Intel Xeon E5-2650 v2	AMD Opteron A1170*
#Cores x #Sock	8 x 2	8 x 1
Clock	2.6 GHz	2.0 GHz
Linux Kernel	3.10.0-327.36.3.el7	4.14.0-115.2.2.el7a
GCC	4.8.5 20150623	

*ARM Cortex-A57 based on ARMv8-A

TABLE III
CONTEXT SWITCH AND LOAD TLS

	Wallaby		Albireo
	Time [Sec]	Cycles	Time [Sec]
Context Sw.	3.34E-8	86	2.45E-8
Load TLS	1.09E-7	284	2.50E-9

a system-call must be called on Wallaby. On AArch64 this time is only a few nanosecond and much faster than that of x86_64.

Table IV shows the times of yielding two threads. The times in this table is normalized to the times of one yield. On ULP-PiP, yielding two ULPs and the time should be the same with sum of the times of context switching and loading TLS register shown in Table III ideally. The column titled as "`sched_yield()` on 1 core" means that two PThreads running and yielding on one CPU core, and another one titled as "`sched_yield()` on 2 cores" means that two PThreads bound to different CPU cores. When the two PThreads running on one core, the call of `sched_yield()`, more precisely `pthread_yield()`, results in actual context switching between the threads. Whereas the case running on two cores, the yield system-call results in doing nothing since there is no other threads running on the same core. Thus, yielding time of two threads on one core takes longer time than that of the two core case because of the context switch overhead. Comparing the x86_64 case of ULP-PiP and `sched_yield()` on two cores, the `sched_yield()` is faster than that of ULP-PiP. This is because the slow TLS loading is involved on x86_64.

TABLE IV
YIELDING TIME (2 ULPS OR PTHREADS)

	Wallaby		Albireo
	Time [Sec]	Cycles	Time [Sec]
ULP-PiP yield	1.50E-7	387	1.20E-7
`sched_yield()` on 1 core	2.66E-7	-	1.22E-6
`sched_yield()` on 2 cores	7.79E-8	-	3.48.E-7

TABLE V
TIME OF `getpid()`

	Wallaby		Albireo
	Time [Sec]	Cycles	Time [Sec]
Linux	6.71E-8	174	3.85E-7
ULP-PiP: BUSYWAIT	1.33E-6	3452	2.71E-6
ULP-PiP: BLOCKING	2.91E-6	6172	4.48E-6

C. `getpid()`

Here, The time to call the `getpid()` system-call is measured. Since the `getpid()` system-call is very light, the overhead of the `couple()` and `decouple()` can be measured. Table V shows the times of `getpid()` and the time of `getpid()` enclosed by `couple()` and `decouple()` in ULP-PiP. There are two ULP-PiP's cases, one for KC to busy-wait (denoted as "BUSYWAIT") and another for blocked by calling a blocking system-call (denoted as "BLOCKING). As for the blocking system-call in this evaluation, the Linux semaphore (implemented by using `futex`) is used. As shown in this table, the `couple()` and `decouple()` overhead is only few microseconds. This overhead includes four times context switching (as shown in Table I) and two times of loading TLS register. In the ULP-PiP cases, busy-waiting outperforms blocking. This is because there is no system-call involved in busy-waiting.

D. AIO vs. ULP

In this subsection, we compared the performance of ULP-PiP's `couple()` and `decouple()` with the Linux's AIO. I/O operations in this evaluation are; 1) open a file on the `tmpfs` file system to exclude the variation of actual disk access, 2) write one block, and 3) close. Note that the comparison between ULP-PiP and AIO might not be fair, since the series of open-write-close system-calls are done on a KLT while the actual AIO is only for the writing. On ULP-PiP. the whole sequence must be done by a KLT otherwise the system-call consistency is broken. In the Linux's AIO implementation, the first AIO call creates a PThread and the same thread is used in the subsequent AIO calls. Our evaluation programs have a warming up loop followed by a measurement loop. So the PThread creation overhead is not included in this evaluation.

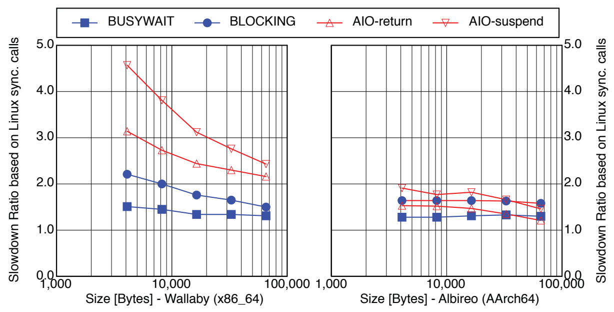


Fig. 7. Slowdown comparison of Open-Write-Close

Figure 7 shows the slowdown ratio based on the time of the Linux's `open()`, `write()`, and `close()` system-calls over the size of the write buffer. Left graph in this figure shows the results running on Wallaby and the right graph shows the results running on Albireo. There are two cases for AIO, one for waiting the done of `aio_write()` by calling the `aio_return()` (denoted as "AIO-return") and another is calling the `aio_suspend()` (denoted as "AIO-suspend"). As described already, calling `aio_return()` is suitable for a ULT to use.

On Wallaby, ULP-PiP outperforms the AIO in all cases. On Albireo, however, ULP-PiP's busy-waiting outperforms AIO slightly if the buffer sizes are less than 32 KiB. In general, the larger the write buffer, the lower the slowdown ratio, if the overhead of ULP-PiP's coupling and decoupling, or the overhead of AIO, is constant over the size of the write buffer. This situation can only be seen on the Wallaby cases.

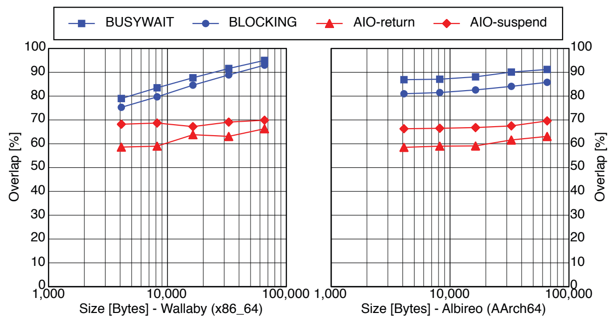


Fig. 8. Comparison of Overlap Ratios

Figure 8 shows the overlap ratio calculated in the way used in the Intel MPI benchmarks [23]. As shown in this figure, the overlap ratios of ULP-PiP are more than 70% on Wallaby and 80% on Albireo whereas the percentages of all AIO cases are less than 70%.

VII. Discussion

One may argue that enclosing system-call(s) by `couple()` and `decouple()` is not practical. Linux supports this kind of system-call wrapper in a several ways, such as GNU wrap (GNU `ld` option) and `LD_PRELOAD`. Even with using one of them, the `couple()` and `decouple()` functions still remain because those functions can enclose a series of system-calls shown in the previous section.

As described above, ULP-PiP can resolve the blocking system-call problem found in ULT. However, this is ineffective when the program is blocked by page faults to create page table entries (minor page faults) and to allocate physical memory pages (major page faults). In system software used for HPC, large (huge) memory pages and/or populated `mmap` are prevalent because they can reduce the number of page faults as well as the number of TLB misses. Therefore in the context of HPC, we believe that handling of page faults at ULP or ULT can be ignored if larger page sizes and/or populated `mmap` are used.

So far, a BLT is created as KLT to create both UC and KC. This is because each UC must have an original KC to preserve system-call consistency. In this sense, our BLT can be categorized as *N:N* model. However, it is not difficult to create a number of ULTs (UCs) having the same original KC in theory and practice. In this case, the UCs having the same original KC access the same information in an OS kernel. This situation is similar to the relation of the conventional process and thread, since threads of a process access the same resources in the OS kernel.

Another argument might be that the ULP requires more OS kernel resources than that of conventional KLT and ULT. While this is true, as already described in equation 2, the number of BLTs or ULPs depends on the number of oversubscribed threads. This resource consumption can be relaxed by having UCs with the same original KC as described above.

As described in the Section VI, the current ULP-PiP implementation can choose the way of idling, busy-waiting or blocking at runtime. As already shown, the busy-waiting introduces less overhead than blocking, however, busy-waiting consumes more power. So the choice of the blocking ways is a trade-off between latency and power. Ideally, one could determine the way of blocking in an automatic way according to the application's behavior or leave this choice as a power nob to control power consumption while maintaining the performance. This is left for our future work.

ULP-PiP has one problem on its system-call consistency. It is signaling. The current implementation uses `fcontext` and it does not save and restore signal masks. So if one tries to send a signal to a UC, then the signal is delivered to the scheduling KC. To avoid this situation, use `ucontext` which saves and restores signal masks. However, to access the signal mask, we have to call a system-call and this adds non-negligible overhead to the context switching. This signal problem cannot be avoided by having a wrapper function of the `kill()` system-call because a signal might be sent via a terminal.

In the scenario in Figure 6, some CPU cores are dedicated for executing system-calls. Recent many-core CPU architectures allow us to do so. Since all system-calls are executed on the dedicated CPU cores, this may result in making the cache footprints by calling system-calls independent from the caches on the CPU cores running user program. This situation is close to the idea of FlexSC [24]. We intend to investigate this effects in the near future.

VIII. Summary

In this paper, we have proposed bi-level threads and user-level processes to resolve the blocking system-call issue and, at the same time, to preserve system-call consistency, which can be crucial in a ULP system. By simply enclosing a system-call or a series of system-calls by the `couple()` and `decouple()` functions provided by the ULP-PiP library, the aforementioned problems can be avoided. Evaluation results show that there is non-negligible overhead introduced by those functions, however, ULP-PiP outperforms the Linux AIO in terms of overhead and overlap ratio.

Is ULP worth investigating? We believe so because users can combine different programs to run simultaneously. This configuration applies to in-situ and multi-physics applications. If applications do not call blocking system-calls frequently, then ULP's context switching overhead is almost equal to the overhead of ULT context switching plus the overhead of loading the TLS register. For CPU architectures that allow accessing the TLS register directly (e.g., AArch64) this overhead is negligible.

References

[1] "Boost C++ libraries," https://www.boost.org.

[2] A. Hori, M. Si, B. Gerofi, M. Takagi, J. Dayal, P. Balaji, and Y. Ishikawa, "Process-in-process: Techniques for practical address-space sharing," in *Proceedings of the 27th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC 18. New York, NY, USA: Association for Computing Machinery, 2018, p. 131143. [Online]. Available: https://doi.org/10.1145/3208040.3208045

[3] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism," in *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, ser. SOSP 91. New York, NY, USA: Association for Computing Machinery, 1991, p. 95109. [Online]. Available: https://doi.org/10.1145/121132.121151

[4] R. Brightwell, K. Pedretti, and T. Hudson, "SMARTMAP: Operating System Support for Efficient Data Sharing among Processes on a Multi-Core Processor," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, ser. SC '08. Piscataway, NJ, USA: IEEE Press, 2008, pp. 25:1–25:12. [Online]. Available: http://dl.acm.org/citation.cfm?id=1413370.1413396

[5] M. Pérache, H. Jourdren, and R. Namyst, "MPC: A Unified Parallel Runtime for Clusters of NUMA Machines," in *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*, ser. Euro-Par'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 78–88. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85451-7_9

[6] K. Kaffes, T. Chong, J. T. Humphries, A. Belay, D. Mazières, and C. Kozyrakis, "Shinjuku: Preemptive Scheduling for Microsecond Scale Latency," in *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI19. USA: USENIX Association, 2019, p. 345359.

[7] A. Shimada, B. Gerofi, A. Hori, and Y. Ishikawa, "Pgas intra-node communication towards many-core architecture," in *In PGAS 2012: 6th Conference on Partitioned Global Address Space Programing Model*, ser. PGAS'12, 2012.

[8] K. Ouyang, M. Si, and Z. Chen, "Exploring interprocess work stealing for balanced mpi communication (research poster)." SC19. [Online]. Available: https://sc19.supercomputing.org/proceedings/tech_poster/poster_files/rpost121s2-file2.pdf

[9] A. Shimada, A. Hori, Y. Ishikawa, and P. Balaji, "User-level Process towards Exascale Systems," *IPSJ SIGARC*, vol. 2014, no. 22, pp. 1–7, dec 2014. [Online]. Available: http://ci.nii.ac.jp/naid/110009850784/

[10] K. Shen, H. Tang, and T. Yang, "Adaptive two-level thread management for fast mpi execution on shared memory machines," in *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, ser. SC 99. New York, NY, USA: Association for Computing Machinery, 1999, p. 49es. [Online]. Available: https://doi.org/10.1145/331532.331581

[11] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in *2008 IEEE International Symposium on Parallel and Distributed Processing*, April 2008, pp. 1–8.

[12] J. Nakashima and K. Taura, "Massivethreads: A thread library for high productivity languages," in *Concurrent Objects and Beyond*, 2014.

[13] S. Seo, A. Amer, P. Balaji, C. Bordage, G. Bosilca, A. Brooks, P. Carns, A. Castell, D. Genet, T. Herault, S. Iwasaki, P. Jindal, L. V. Kal, S. Krishnamoorthy, J. Lifflander, H. Lu, E. Meneses, M. Snir, Y. Sun, K. Taura, and P. Beckman, "Argobots: A lightweight low-level threading and tasking framework," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 3, pp. 512–526, March 2018.

[14] A. Castell, A. J. Pea, S. Seo, R. Mayo, P. Balaji, and E. S. Quintana-Ort, "A review of lightweight thread approaches for high performance computing," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Sep. 2016, pp. 471–480.

[15] O. Krieger, M. Auslander, B. Rosenburg, R. W. Wisniewski, J. Xenidis, D. Da Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, and et al., "K42: Building a complete operating system," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 4, p. 133145, Apr. 2006. [Online]. Available: https://doi.org/10.1145/1218063.1217949

[16] V. Danjean and R. Namyst, "Controlling kernel scheduling from user space: An approach to enhancing applications' reactivity to i/o events," in *High Performance Computing - HiPC 2003*, T. M. Pinkston and V. K. Prasanna, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 490–499.

[17] N. J. Williams, "An implementation of scheduler activations on the netbsd operating system," in *Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference*. USA: USENIX Association, 2002, p. 99108.

[18] S. Iwasaki, A. Amer, K. Taura, S. Seo, and P. Balaji, "Bolt: Optimizing openmp parallel regions with user-level threads," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep. 2019, pp. 29–42.

[19] D. T. Stark, R. F. Barrett, R. E. Grant, S. L. Olivier, K. T. Pedretti, and C. T. Vaughan, "Early experiences co-scheduling work and communication tasks for hybrid mpi+x applications," in *Proceedings of the 2014 Workshop on Exascale MPI*, ser. ExaMPI 14. IEEE Press, 2014, p. 919. [Online]. Available: https://doi.org/10.1109/ExaMPI.2014.6

[20] C. Huang, G. Zheng, S. Kumar, and L. V. Kalé, "Performance Evaluation of Adaptive MPI," in *Proceedings of ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming 2006*, Mar. 2006.

[21] Message Passing Interface Forum. (2015) MPI: A Message-Passing Interface Standard Version 3.1. Message Passing Interface Forum. [Online]. Available: http://mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

[22] U. Drepper, "ELF Handling For Thread Local Storage," 2013.

[23] Intel Corporation, "Intel MPI Bennchmarks User Guide," https://software.intel.com/en-us/imb-user-guide-measuring-communication-and-computation-overlap.

[24] L. Soares and M. Stumm, "Flexsc: flexible system call scheduling with exception-less system calls," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8.