An Implementation of User-Level Processes using Address Space Sharing

IPDPS/RADR 2020 (virtual meeting)

May 18, 2020

Atsushi Hori, Balazs Gerofi ,Yutaka Ishikawa
Riken CCS (R-CCS)
JAPAN

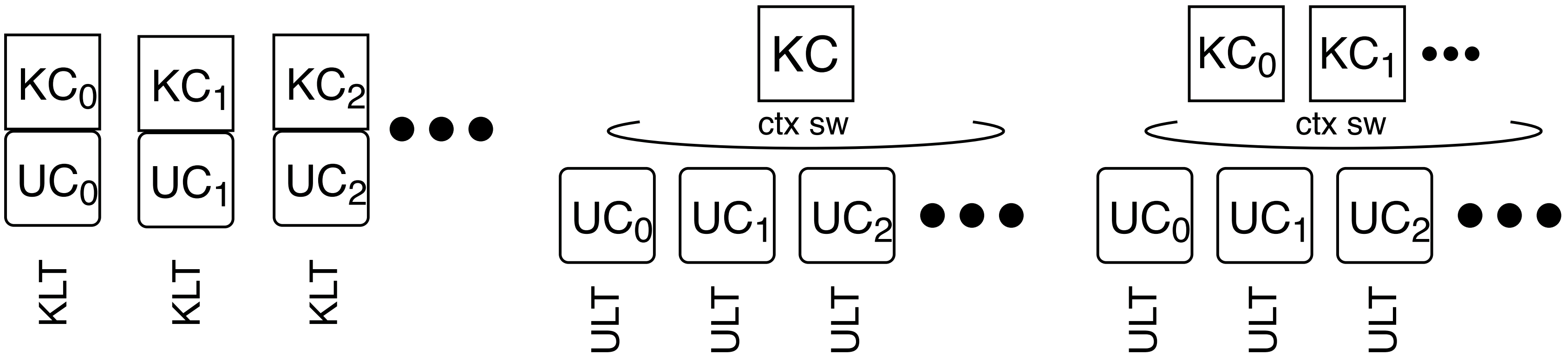
Outline

1. Put an end to the long-term discussion
   - Kernel-Level Thread (KLT) vs. User-Level Thread (ULT)
     - Advantages and disadvantages
2. Challenges
   1. Bi-Level Thread
      - To take the best of the two worlds
      - A ULT can be a KLT and vice versa
      - User-level (thread) context switching
      - Blocking system-call can be called as a kernel-level thread
   2. User-Level Process
      - Combining Bi-Level Thread with Address Space Sharing
        - Process context switching at user-level
3. Evaluation

Bi-Level Thread

Re-thinking Thread Models

- Thread models (1:1, N:1, and M:N)
  - KC: Kernel Context, UC: User Context



(a) Kernel-Level Threads (1:1) (b) User-Level Threads (N:1) (c) User-Level Threads (M:N)

- What if KCs and UCs in 1:1 model can be decoupled and coupled again ?
  - The 1:1 model and M:N (M==N) model can be interchangeable

The Idea of decoupling

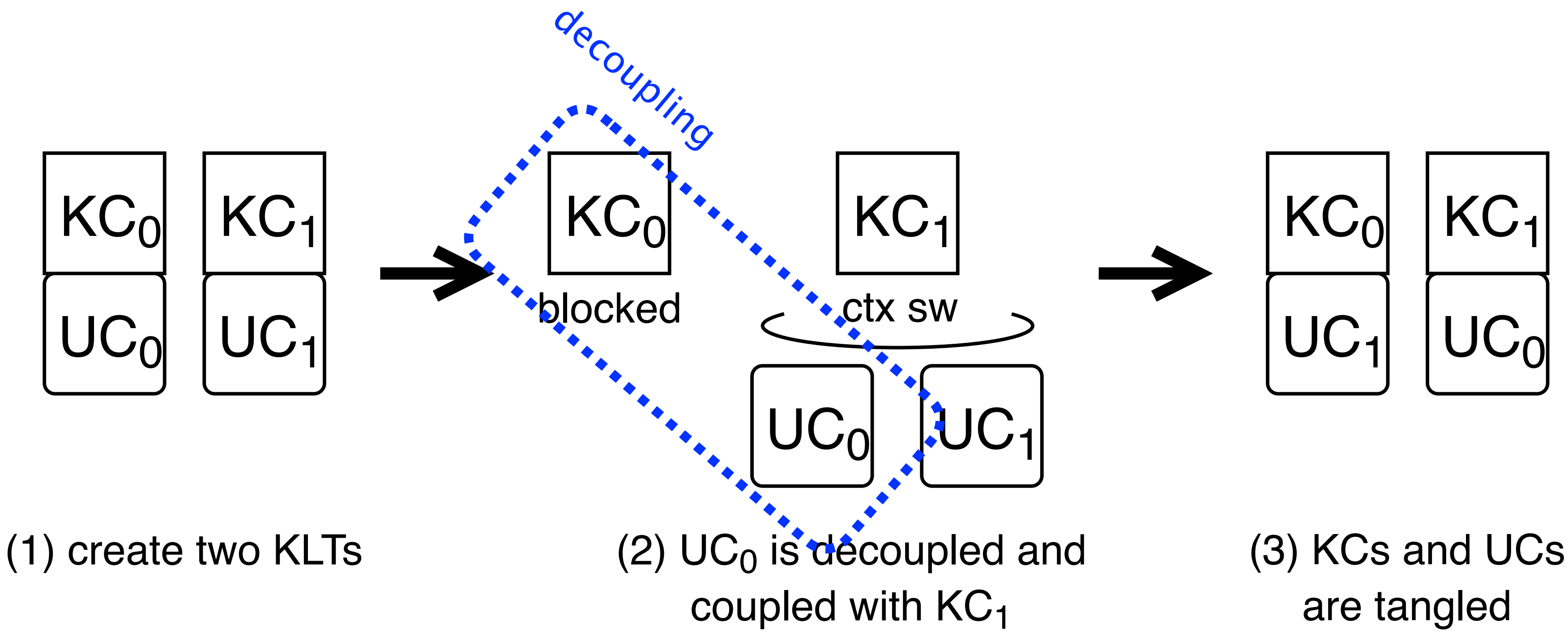


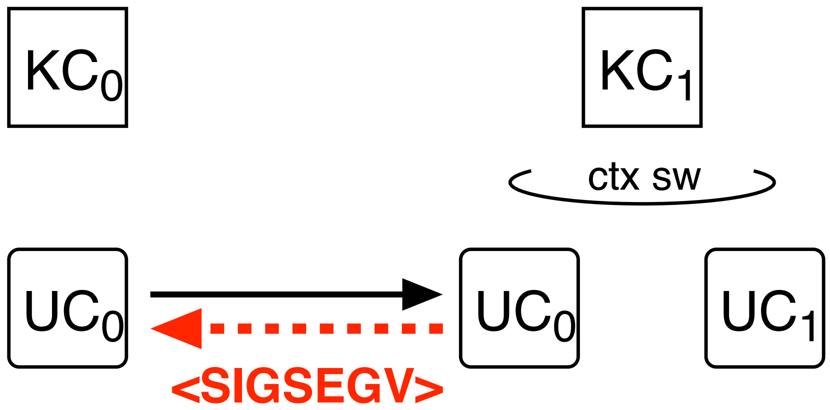
Fig. 2. Coupling and decoupling of UCs and KCs

Decoupling and Coupling

- What if a UC is decoupled from KC ?
  - Decoupled UC can be scheduled by another KC
  - What happens on the decoupled KC ?
    - It has nothing to do (idling or blocked in some way)
  - This is transition for a KLT (Kernel-Level Thread) to be a ULT (User-Level Thread)
- What if the decoupled UC wants to be coupled again ?
  - The idling KC now schedules the UC
  - This is the transition for a ULT to be a KLT
- However, KC must always be associated with a UC
  - A KC cannot be idling without a UC
  - But the UC has to be decoupled so that it can be scheduled by another KC…

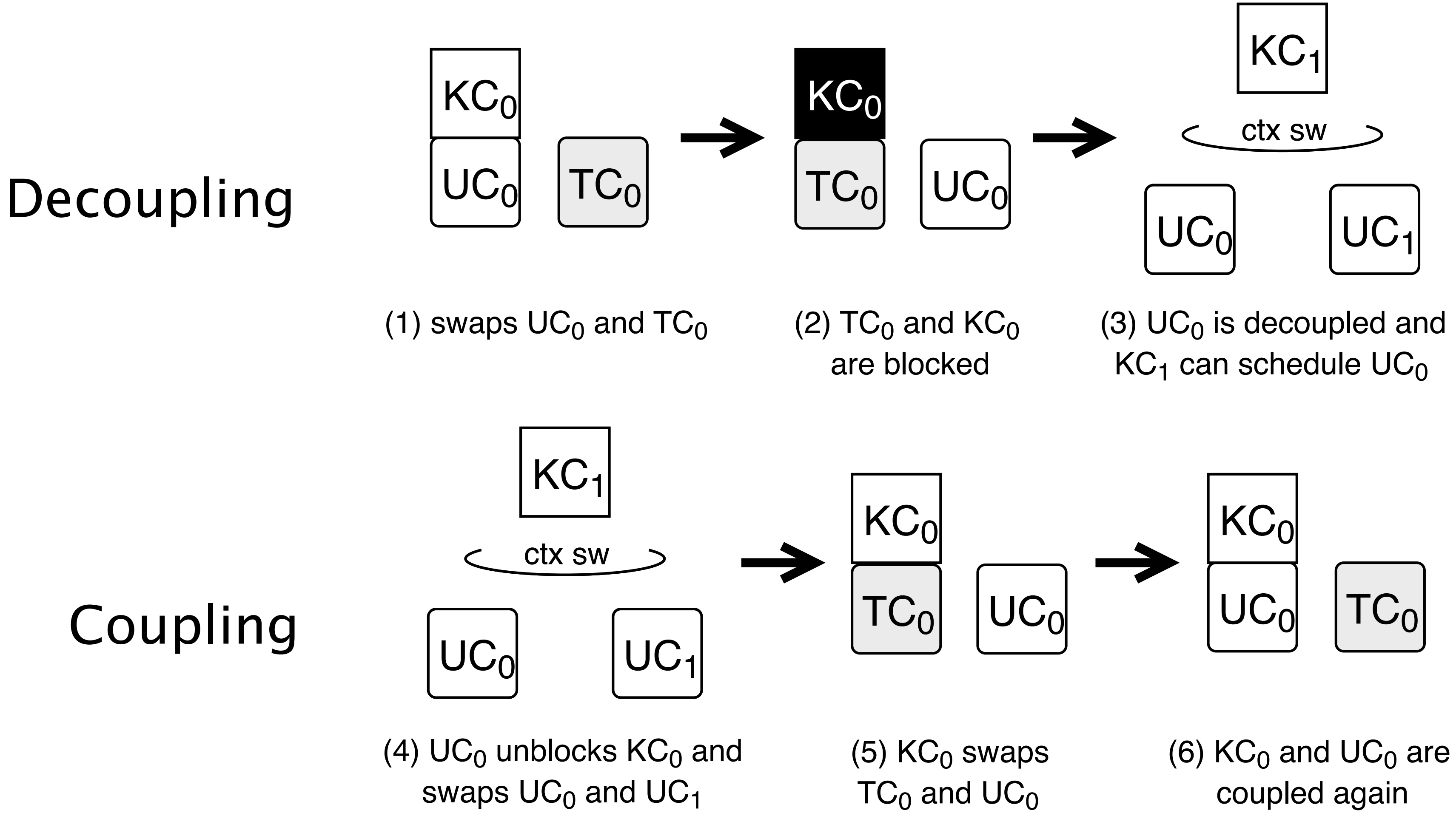
Issue in Decoupling and Coupling

KC_0	KC_1
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);	:



Trampoline Context

- This problem can be resolved by introducing another small context (Trampoline Context, TC)



Resolving Blocking System-call Issue

- Issue
  - When a ULT calls a blocking system-call,
    - the scheduling KC is also blocked, and
    - the other eligible-to-run ULTs have no chance to be scheduled
- Solution by using coupling and decoupling
  - Assumption:
    - A ULT is going to call a blocking system-call
    - The ULT was firstly created with KCo
    - The ULT is already decoupled and scheduled by the KCs

1. before the ULT calls the system-call, it is coupled with KCo
2. (KCs can schedule the other ULTs)
3. the ULT becomes a KLT and it calls the system-call
4. after returning from the system-call, it is decoupled so that KCs can schedule it (becoming ULT again)

User-Level Process

Address Space Sharing

- What is Address Space Sharing (ASS) technique ?
  - “Processes” share the same address space
  - Here “process” is defined as an execution entity having privatized static variables, and
    ASS “processes” may be derived from different programs
    - threads share all static variables, and
      threads are derived from the same program
  - ASS is different from POSIX shared memory (PSM)
    - ASS share the whole address space (and a page table)
    - PSM shares only some specific memory pages
- Process-in-Process (PiP)
  - Pure user-level implementation of ASS

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.

BLT + ASS = User-Level Process (ULP)

- ASS allows for a process to context-switch one to the other at user-level => Fast context switching
- The difference between ULT and ULP
  - Threads share most OS kernel resources while processes do not
  - Example: getpid()
    - threads have the same PID
    - each process has its own unique PID
- In ULP
  - System-call consistency can be preserved
    - by using the same (de)coupling technique in BLT
  - Thread Local Storage (TLS) must also be switched when switching contexts
    - In most ULT implementations, TLS switching is ignored

Evaluation

Evaluation Results

- Machines: Wallaby – x86_64, Albireo – AArch64

BUSYWAIT/BLOCKING: Idling ways of KC w/ TC

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

On x86_64 CPUs a system-call is required to switch TLS

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

In ULP-PiP cases, getpid() call is wrapped by pip_couple() and pip_decouple()

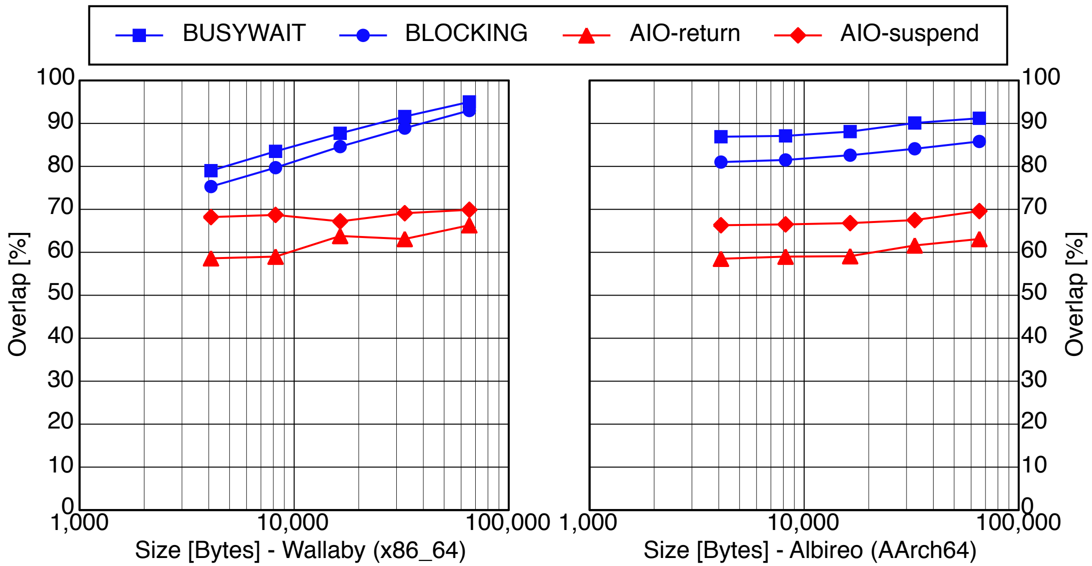


Fig. 8. Comparison of Overlap Ratios

```
ULP-PiP:
  pip_couple();
  open();
  write(Size);
  close();
  pip_decouple();
```

```
AIO-return:
  open();
  aio_write(Size);
  do {
    aio_return();
  } while();
  close();
```

```
AIO-suspend:
  open();
  aio_write(Size);
  aio_suspend();
  close();
```

Summary

ULP-PiP will be available at
https://github.com/RIKEN-SysSoft

- Proposing
  - Bi-Level Thread (BLT)
    - Decoupling and coupling UC and KC
    - Trampoline Context to block decoupled KC
    - Able to handle blocking system-calls effectively
  - User-Level Process (ULP) by using Address Space Sharing
    - Switching Thread Local Storage (TLS)
    - System-call consistency
  - Coupling and decoupling can be applied to
    - resolve the blocking system-call issue, and
    - preserve system-call consistency in ULP
- Evaluation (ULP vs. AIO)
  - Coupling and decoupling scheme of ULP-PiP outperforms AIO