Application Fault Tolerance Using Continuous Checkpoint/Restart

Tomoki Sekiyama

Linux Technology Center

Yokohama Research Laboratory

Hitachi Ltd.

Outline

1. Overview of Application Fault Tolerance and Continuous Checkpoint/Restart
2. User-space Implementation of Continuous Checkpoint/Restart
3. Kernel-space tracking of Dirty pages

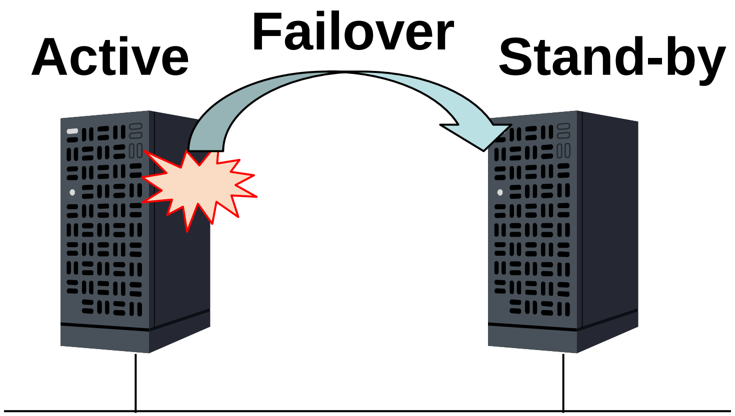
1-1. Overview of Fault Tolerance

- Availability of information systems is important especially in
  - Mission-critical systems
    (e.g. Banking, stock exchange)
  - Control systems
    (e.g. Factory-automation, infrastructures)

- Real-time performance is also required.
  - Even short delays can cause large accidents.

1-2. Redundant Configuration

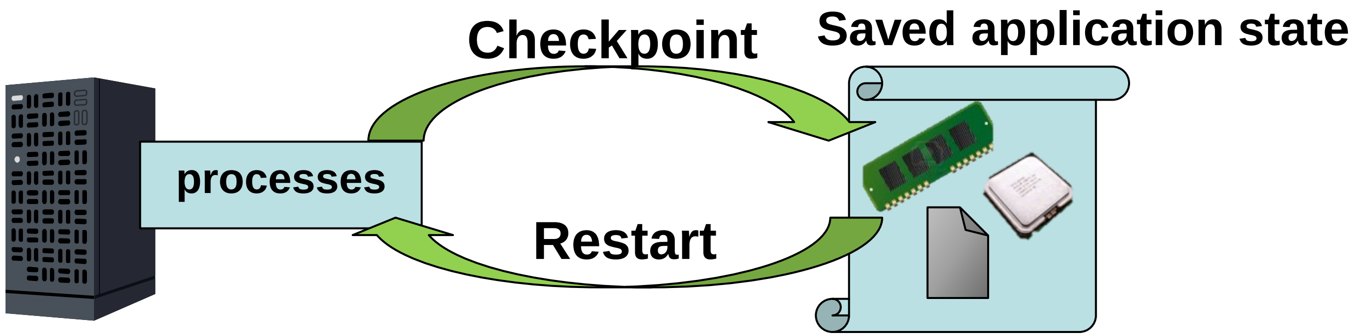
- Redundant configuration is often used to improve availability



- On-memory data are lost on the server failure
- To implement non-stop failover, data coherency must always be achieved between active/stand-by servers.
  - Applications must implement data synchronization
    - Difficult to eliminate bugs completely

1-3. Checkpoint/Restart Overview

- Checkpoint/Restart (C/R) is an application independent methods to implement high availability



  - **Checkpoint**:
    - Save an application state (memory, registers, file descriptors, …) at pre-determined points.
  - **Restart**
    - Resume an application execution using the last saved state, on the same machine, or on another machine.
- Similar to snapshot of VM, but done at application level only (OS internal state is not saved/resumed).

1-4. Adopting C/R to RT systems

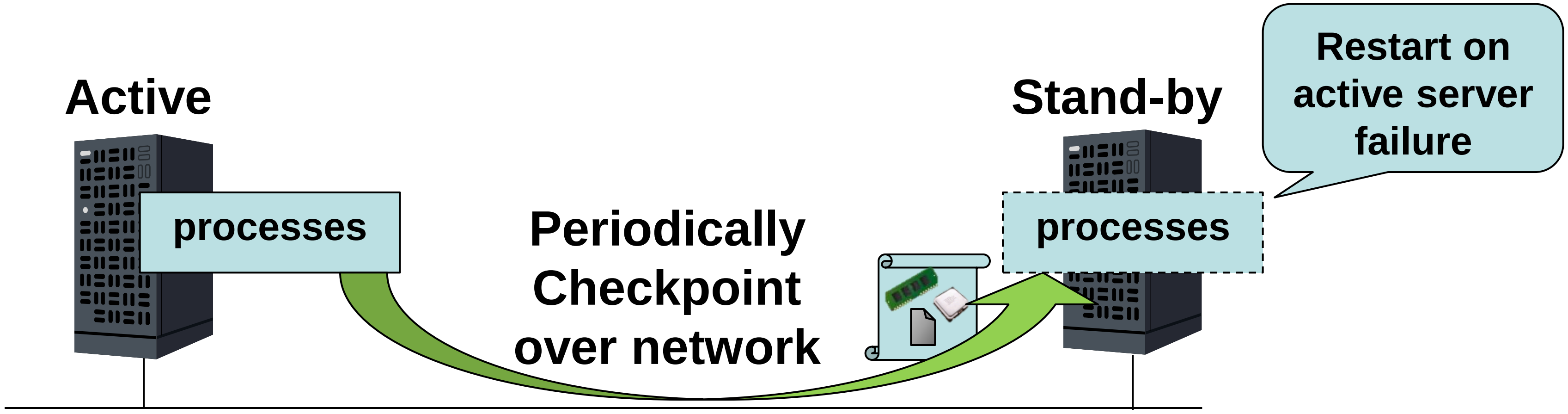
- DMTCP: user-space implementation of C/R
  - Realize C/R transparently for various applications, including MPI-based HPC applications
  - Save application processes states to a file

- For HPC purposes, usually the application is restarted after repair of the server; saving states to a file is reasonable.
- **To adopt C/R to real-time control systems, shorter down time is required (e.g. 1sec)**

⇒ Continuous C/R is introduced

1-5. Continuous C/R

- To realize fault tolerance for real-time systems, continuous C/R is introduced
  - Application must ensure periodical state transfer of to another stand-by server (e.g. every second)
    - Stand-by server update the process image, but doesn't restart
  - When server failure is detected, the stand-by server restart the applications



1-6. Requirement of Continuous C/R

- Application down time must be as short as possible
- Reduce overhead of checkpoint processing

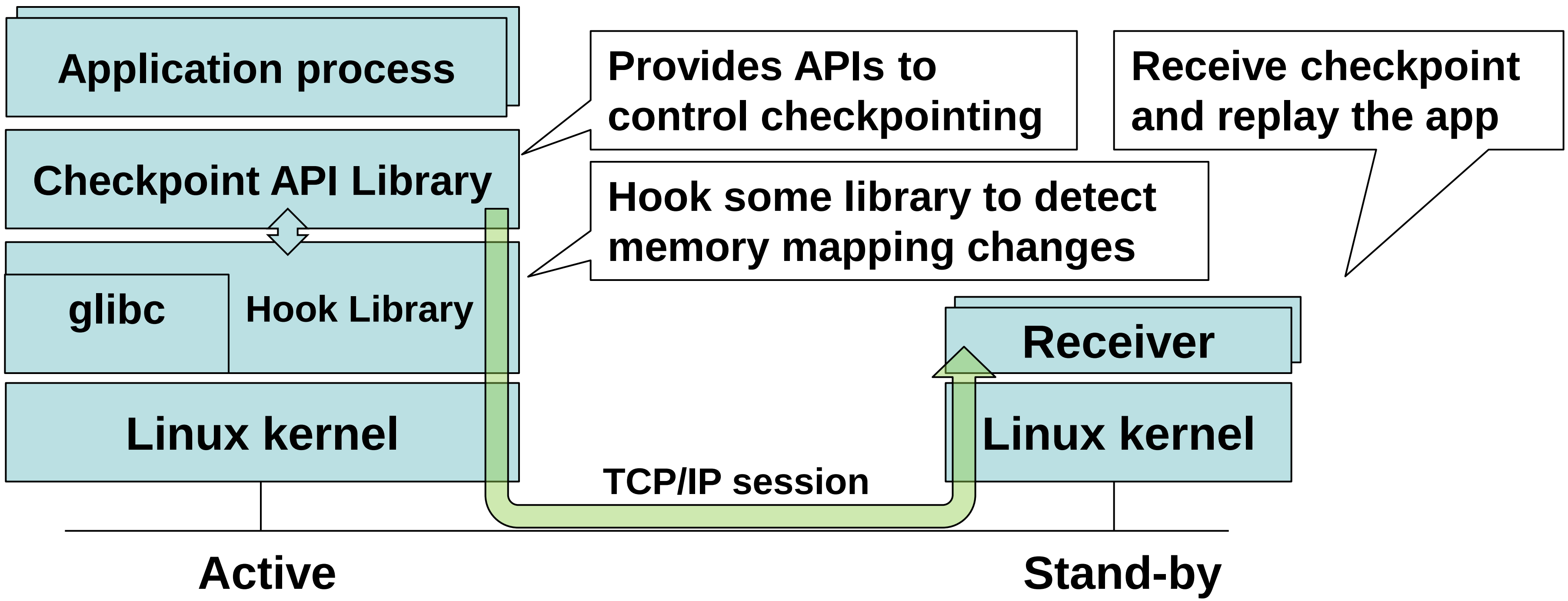
- It takes too much time to transfer entire memory at every checkpoint. Transfer size must be reduced…
  ⇒ Incremental transfer of modified memory since last checkpoint
  ⇒ **Dirty**(=not synchronized) **memory detection** is needed

- Applications have hot spot (frequently rewritten areas) and cold spot (rarely rewritten areas)
  Cold spots can be transferred without wait until checkpoint
  ⇒ **Background transfer** of memory image is needed

Contents

1. Overview of Application Fault Tolerance and Checkpoint/Restart
2. User-space Implementation of Checkpoint/Restart
3. Dirty pages tracking in Kernel-space

2-1. Software Stack



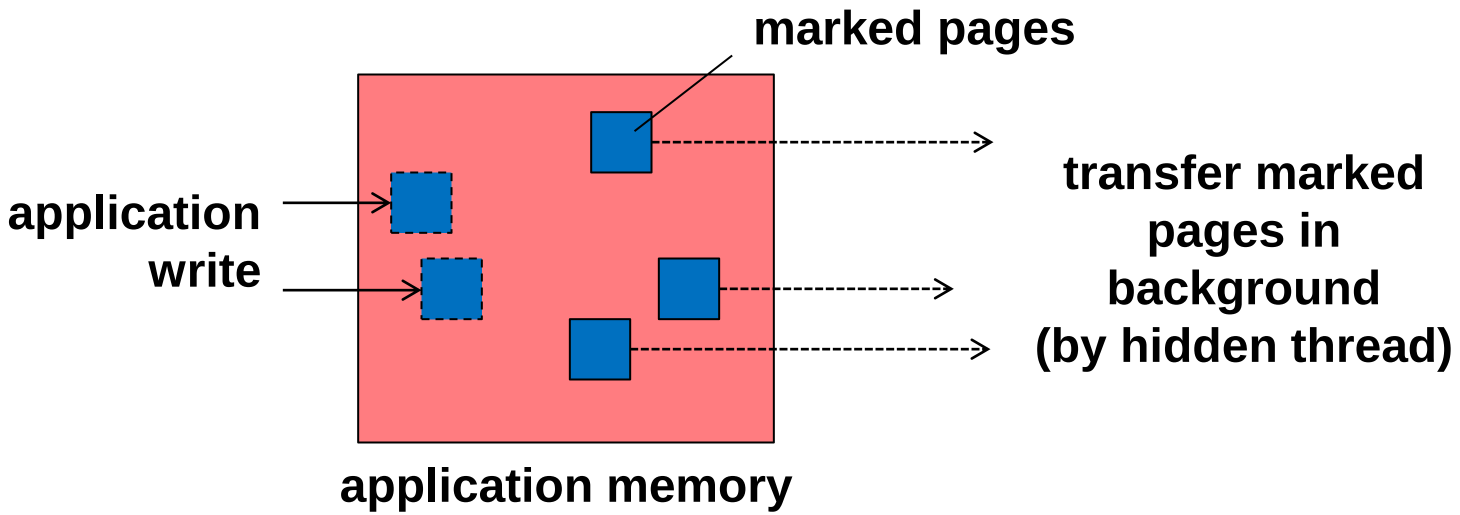
- Current design requires modifications to application
  - Application needs to link the checkpoint API library, and specify checkpoint
    - Modification can be avoided by LD_PRELOAD and controlling the checkpoint from external coordinator, like DMTCP.
  - Some functions in glibc (mmap, munmap, brk, …) are hooked

2-2. Checkpointing Memory Image

- Which memory areas should be synchronized to restart?
  - Anonymous pages (stack, heap, etc.)
    - Need synchronization
  - File-mapped pages
    - Private + writable (application/libraries' .data section, etc.)
      - Need synchronization
    - Private + read-only (application binary, libraries, data files, etc.)
      - Don't need if the same file is on stand-by server
      - Special care is needed for mprotect(2)-ed areas after modification (/proc/<pid>/smaps gives hints for this)
    - Shared
      - Only need synchronized once

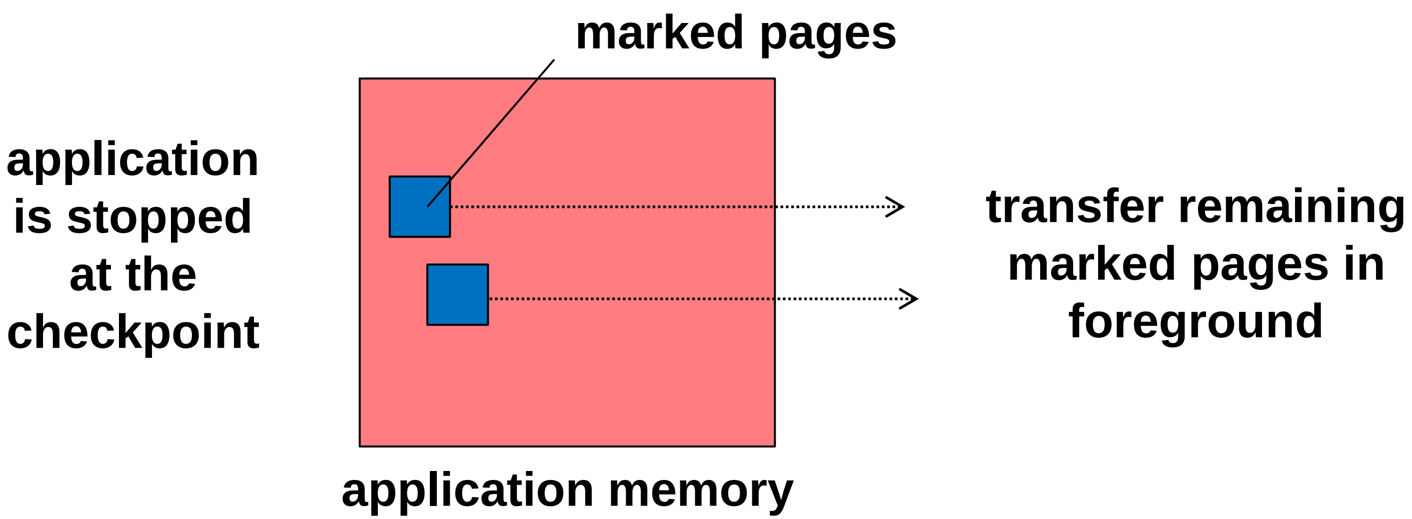
2-4. Background transfer

- In addition, transferring memory image is split into 2-phases to shorten application down time
  - Phase 1: Background transfer
    - Started on application launch
    - Transfer memory image while application is running
    - Asynchronous = Inconsistent
  - Memory mapping modifications (mmap, munmap, brk…) are detected by glibc hook



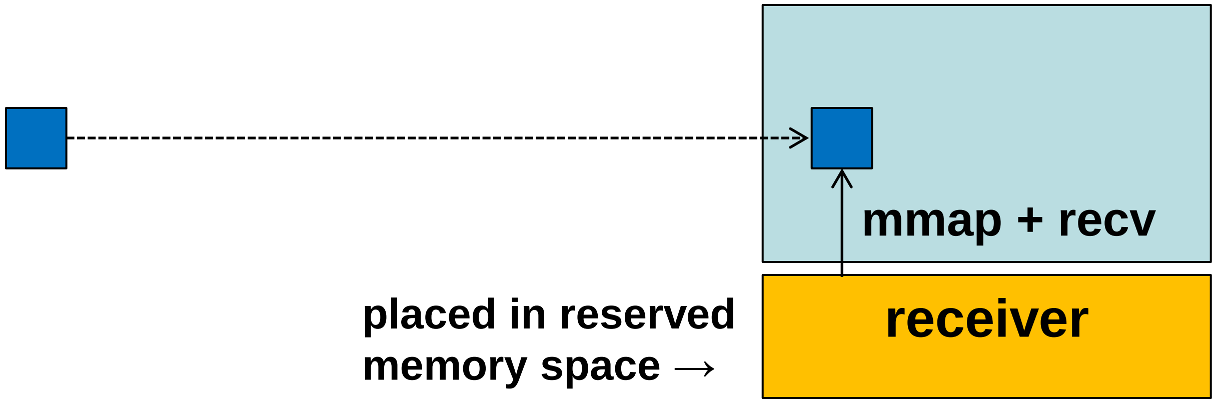
2-4. Background transfer

- In addition, transferring memory image is split into 2-phases to shorten application down time
  - Phase 2: Foreground transfer
    - Stop application at the checkpoint
    - Transfer consistent memory image
  - Memory mapping information, registers values (obtained by setjmp), file descriptors information are also transferred.



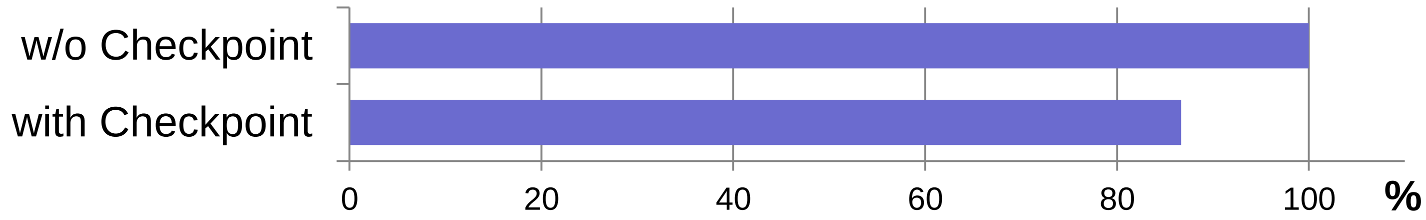
2-5. Receiver

- Transferred memory image is replayed by the receiver
  - Receiver runs in reserved virtual memory address
  - Not linked to any other libraries
  - mmap(2) memory and recv(2) data to original address
  - CoW is used to keep consistent memory image
- When active server failure is detected, restart the application
  - Reopen file descriptors
  - Recover registers (using longjmp) to restart the application



2-6. Performance Evaluation

- Overhead of dirty page detection largely depends on the application memory access pattern
  - First write access causes de-protection
  - Following accesses can be done without overhead
- Example:
  - Application rewrites 100MB memory between checkpoints
  - Checkpoint every 3 seconds



  - SIGSEGV handler takes 10μs* for each page
  - ~300ms is consumed to mark pages for each 3 seconds period
  - Application down time at the checkpoint is 100ms (foreground transfer size is about 5-10MB)

* tested on Intel Xeon 3520 processor

Contents

1. Overview of Application Fault Tolerance and Checkpoint/Restart
2. User-space Implementation of Checkpoint/Restart
3. Dirty pages tracking in Kernel-space

3-1. Kernel-space Dirty Page Tracking

- Dirty page tracking using SIGSEGV is inefficient
- CPU set modified bits in the page table on write
  - Can detect dirty pages without overhead

- Microsoft Windows has APIs to track modified pages (for profiling, debugging, and GC hinting)
  - `ResetWriteWatch()` : Begin modified page tracking
  - `GetWriteWatch()` : Get modified pages since last reset

- FreeBSD mincore systemcall
  - `int mincore(void *addr, size_t len, char *vec);` returns presence / referenced / modified bits of pages into vec
  - However, no way to clear modified bits

- How about Linux?

3-2. Dirty page tracking APIs

- Linux mincore(2) system call
  - `int mincore(void *addr, size_t length, unsigned char *vec);`
    - Only returns whether pages are resident in memory (Based on present bit in page table and PageUptodate)

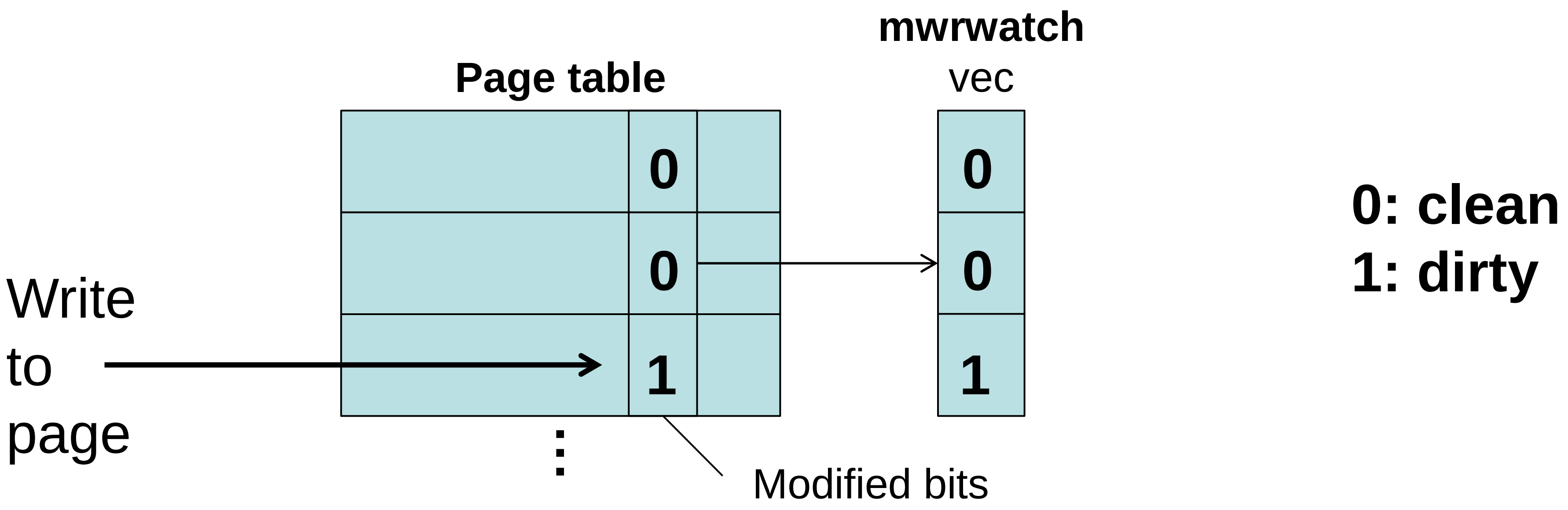
- We added similar interface system call to track dirty pages
  - `int mwrwatch(void *addr, size_t length, unsigned char *vec);`
    - Returns following values into vec
      - WATCH_CLEAN: the page is NOT updated since last call
      - WATCH_DIRTY: the page is updated since last call or it is the first time call
      - WATCH_UNMAPPED: the page is not present

      * for currently unsupported pages type:
      - WATCH_FILE: the page is file-backed
    - If vec == NULL, it just resets the modified bits (to begin tracking)

3-3. Implementation of mwrwatch

- In mwrwatch:
  1. Scan modified bits in page table, set vec to WATCH_CLEAN / DIRTY
     (for the first time call, set DIRTY for every page)
  2. If modified bits are set, clear them (and set dirty flag in page struct)

- If application writes to the memory, the dirty bit is set
- At the next call of mwrwatch, the modified pages are marked in vec



3-4. Implementation Details

- **Current implementation doesn't support swapped out pages nor shared pages (by fork(2) or KSM)…**

1. Write-lock mmap_sem
2. Check vm_area_struct which corresponds to the specified address range to determine if the pages type is supported or not
3. Scan the page table entries : *

   3-1. Clear PTE entry and flush TLB to block access to the memory

   3-2. If dirty bit is set :

   clear dirty bit; call set_page_dirty(); **

   vec[i] = WATCH_DIRTY;

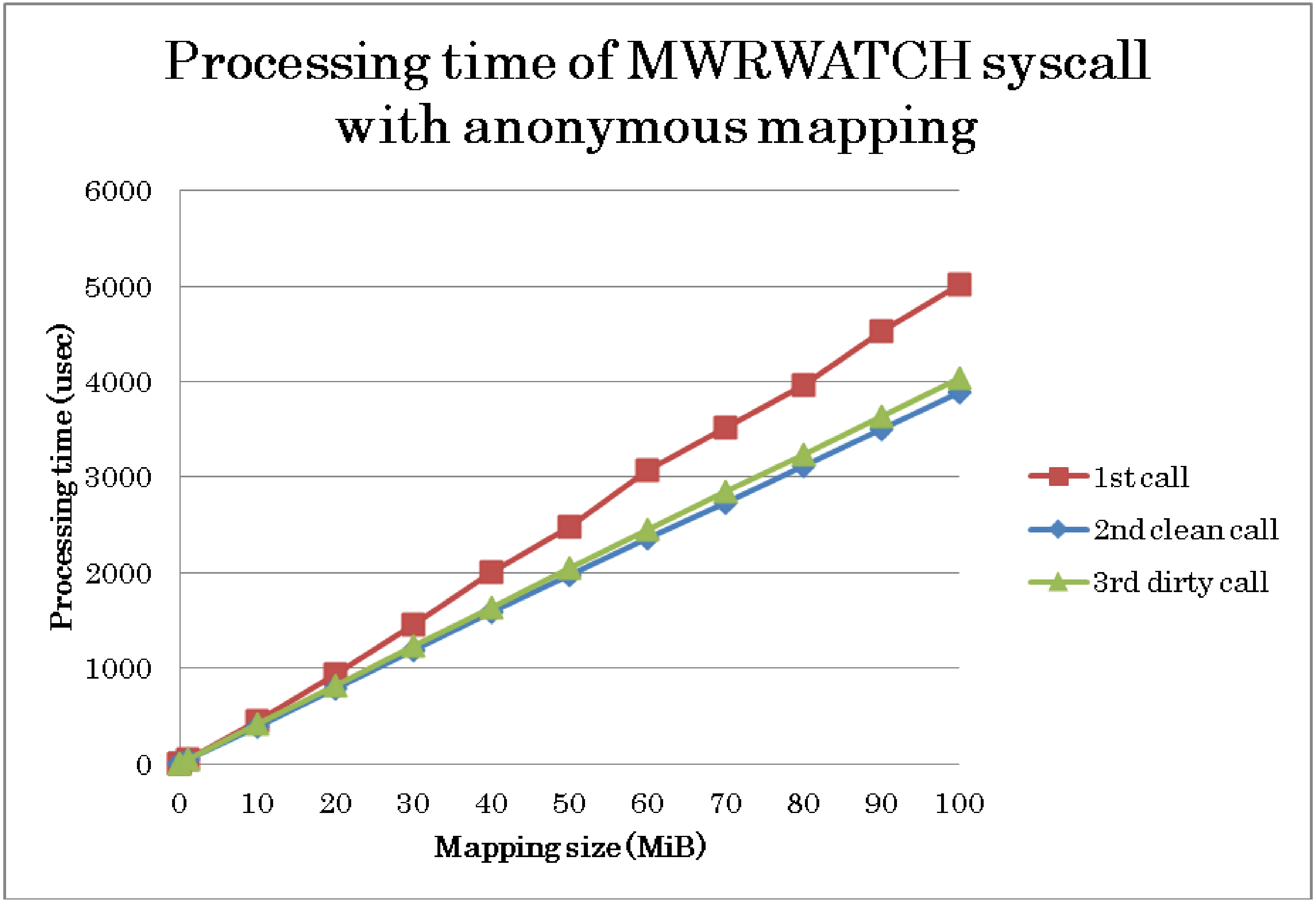
   Else: vec[i] = WATCH_CLEAN;

   3-3. Revert PTE entry
4. Unlock mmap_sem

* When transparent huge page is used, split it into 4KB pages for later tracking

** Set dirty flags in struct page to notice mm subsystem

3-5. Performance of mwrwatch

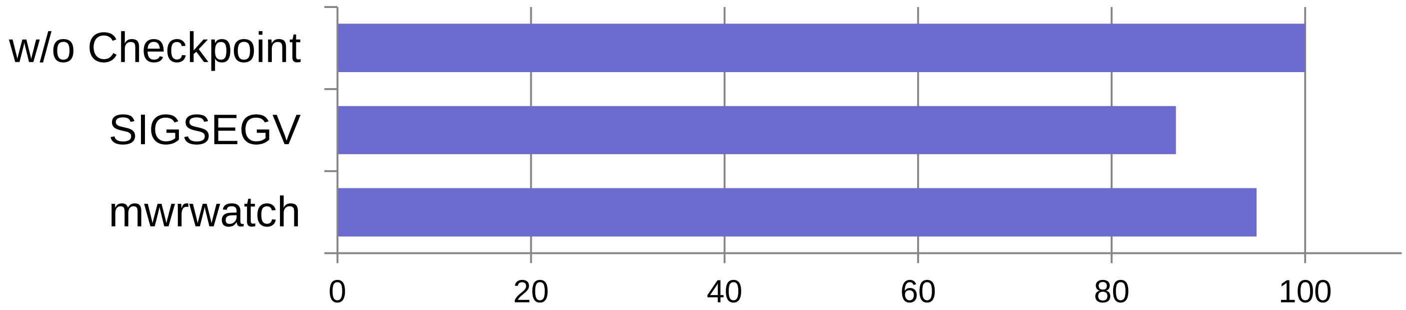


3-6. Performance analysis

- 1st call: 50μs / MB ← clear PTE dirty bits + SetPageDirty
- 2nd clean call: 38μs / MB ← no operations
- 3rd dirty call: 40μs / MB ← clear PTE dirty bits

c.f. SIGSEGV: 3000μs/MB

– Example:
  - Application rewrites 100MB memory between checkpoints
  - Checkpoint every 3 seconds



- Almost no dirty page marking overhead while the app is running
- Application down time at the checkpoint is 100ms (~5% overhead)
  - Lots of vm scans are needed even when there is no dirty pages
  - If the app has many processes (large vm), it takes much time

Conclusion

- Continuous Checkpoint/Restart is an application independent method of application fault tolerance
- Dirty page detection can be implemented in user-space using mprotect
  - But overhead of SIGSEGV handling is large and unpredictable…
- By adding mwrwatch system call, overhead can be eliminated
- To-do
  - Upstreaming modified page tracking mechanism

    Modified page tracking is also useful for debugging, profiling,GC hinting, etc.
    - interface may need brush-ups for such purposes
    - File-backed / shared / swapped-out pages should be supported.

Thank you!

Questions?

Copyrights and Trademarks Notices

- Linux is a registered trademark of Linus Torvalds.
- Microsoft and Windows are trademarks of Microsoft Corporation.
- FreeBSD is a registered trademark of The FreeBSD Foundation.
- All other trademarks and copyrights are the property of their respective owners.

HITACHI
Inspire the Next