Cache-based checkpointing mechanism for High-Availability in Safety Critical Systems

Antonio Savino, Antonio Sposito

***Abstract*—System availability is the percentage of time in which an operating system is not down. Implementing fast checkpointing mechanisms within hypervisors is one of the biggest challenges in safety critical systems. However, using traditional checkpointing techniques based on storing data on disk can cause excessive latency and system overhead, limiting application performance and scalability. To overcome this problem, cache-based checkpointing mechanisms has recently been proposed in order to reduce the system downtime. We aim to provide a new faster recovery mechanism based on cache predictable behaviours safety critical embedded systems.**

I. Introduction

Checkpointing is a mechanism used in computer systems to ensure system availability and reliability. It involves saving the current state of the system at a specific point in time, so that it can be restored in case of failure or error. This process is carried out by dedicated software, called a checkpointing manager, which saves important data and information about the system. There are several types of checkpointing, including: periodic checkpointing, in which checkpointing is performed at regular time intervals; incremental checkpointing, in which only the changes made to the system are saved in a checkpoint; application-level or system-level checkpointing in which, respectively, only application or system data are saved.

While useful, checkpointing mechanisms in safety-critical systems can suffer from issues such as overhead (requiring additional resources) and latency (limited by disk speed), which can be particularly problematic in real-time applications. With embedded systems requiring an increasingly number of security and real-time features, virtualization has emerged as a solution. However, many general-purpose hypervisors (such as Jailhouse and Bao) lack support for a checkpointing mechanism, which can improve system safety and availability [1].

–What if we wanted to introduce in these hypervisors a fast cache-based checkpointing mechanism with a low latency and overhead in order to reduce the downtime of the system? Our work focused on studying the feasibility of this mechanism exploiting the cache space efficiently. In particular:

1) We modified Bao code to support a page permission restoration mechanism in memory.
2) To optimize the access to the same page in the future, we have also explored the feasibility of pre-fetching the page within the restoration mechanism.
3) After testing both modifications in emulation using Qemu, we tested them on the Zcu104 board.

II. Background

A. Bao

Bao (from Mandarin Chinese “b ˇaohù”, meaning “to protect”) is a security and safety-oriented, lightweight bare-metal hypervisor. Designed for MCSs, it strongly focuses on isolation for fault-containment and real-time behavior. Its implementation comprises only a minimal, thin-layer of privileged software leveraging ISA (Instruction Set Architecture) virtualization support to implement the static partitioning hypervisor architecture: resources are statically partitioned and assigned at VM instantiation time; memory is statically assigned using 2-stage translation; IO is pass-through only; virtual interrupts are directly mapped to physical ones; and it implements a 1-1 mapping of virtual to physical CPUs, with no need for a scheduler [2]. Bao relies on a configuration file to implement the hardware static partitioning, this file specifies the number of virtual machines, the number of CPUs and the memory regions to be assigned to each of them. After reading the configuration file, Bao initializes the CPUs, memory regions and interrupts. Then it assigns to each virtual machine a specific number of pages that will be used for the memory mapping, the device mapping and for inter-process communications.

B. Cortex-A53 Processor

In our experiments, both in emulation and on board, we dealt with the Arm Cortex-A53 processor, an extremely power efficient processor capable of supporting 32-bit and 64-bit bit code, which has from one up to four

cores in a single cluster. From now on, all considerations on the architecture implemented by Bao will be related to the Armv8-a architecture and, specifically, to the just mentioned Cortex-A53. In table I are reported the most "juicy" pieces of information about the Cortex-A53 for our experiment

TABLE I: CORTEX-A53 MAIN FEATURES

Release date	July 2014
Typical clock speed	2GHz on 28nm
Cores	1 to 4
Pipeline stages	8
L1 Cache size (Instruction)	8KB to 64 KB
L1 Cache structure (Instruction)	2-way set associative
L1 Cache size (Data)	8KB to 64KB
L1 Cache structure (Data)	4-way set associative
L2 Cache	Optional
L2 Cache size	128KB to 2MB
L2 Cache structure	16-way set associative
Main TLB entries	512

The Cortex-A53 processor is normally implemented with two levels of cache: a small L1 Instruction and Data cache for each core and a larger, unified L2/LLC cache, which is shared between multiple cores in a cluster. Cache memory provide a fast way to access recently used data, but Micro-architectural contention at shared last-level caches (LLCs) and other structures still allows for interference between guest partitions. BAO implements a page coloring mechanism enabling LLC cache partitioning; this solves the interference among guests but introduces memory waste and fragmentation.

C. Memory management

As stated before, Bao uses a 2-stage translation process to ensure that a VM can only see the resources that are allocated to it, and not the resources that are allocated to other VMs.

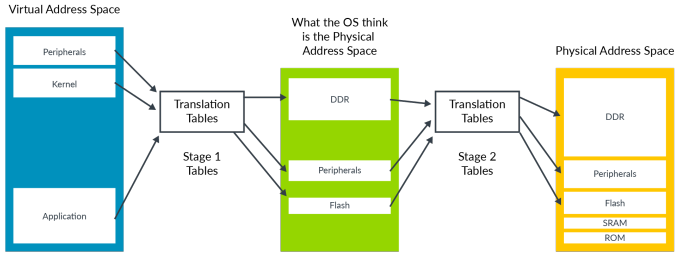

Fig. 1. 2-stage translation [3]

Figure 1 shows how the OS uses a set of translations tables that map from a virtual address (VA) space to what it thinks is his physical address space. The hypervisor uses a second set of translation tables to make a second translation from the Intermediate Physical Address (IPA) to the real Physical Address (PA).

Each translation table is an array of Page Table Entries (PTE) and each of them holds the mapping between a virtual address and a physical one, as well as other information like the access and share permissions and execution flags. When a memory access occurs the Memory Management Unit (MMU) checks the Translation Lookaside Buffer (TLB) which cache all recently used translations; if the MMU does not find a recently cached translation, the table walk unit reads the appropriate table entry, or entries, from memory. As showed in figure 2, each entry can lead to another entry in what is called Multilevel translation: practically the page table is divided into an hierarchy of tables.

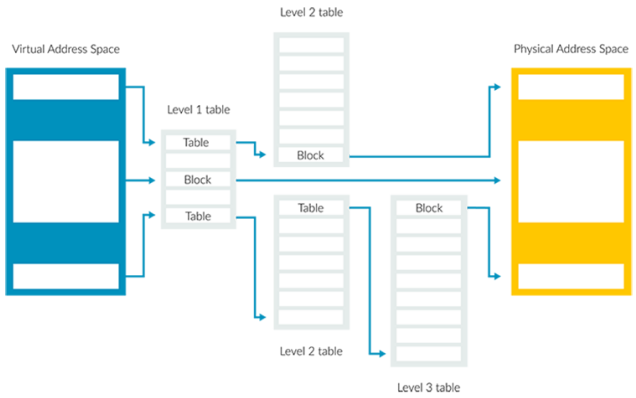

Fig. 2. Multilevel translation [4]

D. GDB

In order to understand the precise register that has to be observed/modified in our experiment, we found it very useful to use gdb-multiarch, in addition to consulting the documentation on the ARM DEVELOPER website. GDB is a source-level debugger capable of setting breakpoints in a program at specified lines, displaying variable values and showing which part of the code is in use at any given time. We only used this debugger in emulation, allowed us to connect to the remote machine using 'target remote:1234' to discover how the code was executed.

III. OBJECTIVES

During the initialization phase, Bao assigns read and write permissions (and other flags) to each page, allowing VMs to access memory for any operation. We caused a data abort exception fault by setting the VM permissions for a page to write only, after the VM

attempts a read operation the exception is raised and the hypervisor has to intervene. Since Bao does not support an handler for this fault, we patched the code by adding a simple handler that calculates the physical address of the page, finds the related page table entry, and restores normal read and write permissions. To determine the impact of such operation, we measured the time required to execute the VM exit, the time needed to execute the handler we implemented and, finally, the time needed to execute the VM entry; then we compared this data with the time required to do a memory read operation from both disk and cache. By comparing these times, we were able to quantify the impact such a mechanism would have if it was implemented in Bao. To optimize subsequent memory accesses to the page that caused the fault a pre-fetching of the page could be implemented within the handler itself; unfortunately, we have not been able to add this feature to our handler.

IV. Our Patch

In order to understand the code of our patch we will explain first how Bao assigns page permissions (section IV-A), second, we will illustrate the baremetal guest code executed the VM managed by the hypervisor (section IV-B) and finally we will present the code of the handler that restore access permissions (section IV-C).

A. Memory mapping

Bao uses a function called *mem_map* (located in the *mem.c* file) to map the allocated physical pages with the corresponding virtual pages filling in this way the translation tables.

```
int mem_map(addr_space_t *as, void *va,
ppages_t *ppages, size_t n, uint64_t
flags)
```

Within this function, after several checks, the *pte_set* function is called to set the Page Table Entries (PTEs) at each of the 3 translation levels (as explained in the section II-C); we are interested in a single PTE which refers to a page at translation level 3. The *pte_set* function makes simply an OR between the address of the page address **addr**, the page type **type**, and the access permission flags **flags** and saves this information in the address pointed by **pte**.

```
static inline void pte_set(pte_t* pte,
uint64_t addr, uint64_t type,
uint64_t flags){
*pte = (addr & PTE_ADDR_MSK) |
       (type & PTE_TYPE_MSK) |
       (flags & PTE_FLAGS_MSK);
}
```

We modified the *mem_map* function: when the physical address (**0x401a7000**) of the page we want to use in the experiment is encountered only write permissions are assigned to it, while the default permissions (write and read) are set for the other pages.

```
int mem_map(...){
[...]
if (paddr == 0x401a7000){
uint64_t wo_flags=PTE_VM_FLAGS_WO;

pte_set(pte, paddr,
        pt_pte_type(&as->pt, lvl),
        wo_flags);
}else{
pte_set(pte, paddr,
        pt_pte_type(&as->pt, lvl),
        flags);
}
[...]
}
```

This function was also used to find the IPA (**0x1a2000**) corresponding to the physical address of our page.

B. VM Baremetal Code

The code run by the VM is located into a c file called *main.c*. We added a write operation into the page we used for our experiment. This operation is the only one allowed by the previously set permissions.

```
int* ptrA = NULL;
void main (void){
[...]
    ptrA = 0x1a2000;
    *ptrA=5;
[...]
}
```

The code also stes up an UART interrupt, we can use the receiving handler (*uart_rx_handler*) to implement the reading operation that will trigger the data abort fault.

```
void uart_rx_handler(){
static int counter = 0;
printf("\n\ncpu%d: %s: %d\n",
```

```
    get_cpuid(),
    __func__, counter++);
int a = 0;
int ptr = 0x1a2000;

MSR(PMCCNTR_EL0, 0);

uint64_t begin = MRS(PMCCNTR_EL0);
asm volatile("ldr %0, [%1] ": "=r"(a)
: "r" (ptr): "memory"); //read
uint64_t bar_bef =  MRS(PMCCNTR_EL0);
asm volatile("dsb sy" ::: "memory");
asm volatile("isb" ::: "memory");
uint64_t bar_aft =  MRS(PMCCNTR_EL0);
printf("a = %x \n", a);
uint64_t end = MRS(PMCCNTR_EL0);

[...]
uart_clear_rxirq();}
```

After printing the number of the CPU that handles the signal, a *LDR* (Load with immediate offset) operation is used to read from the page we are using. Since the page has been initialized without read permissions, a data abort fault is raised and the hypervisor has to intervene to handle it. After the *LDR* operation we placed a Memory Synchronization Barrier (the program waits for all operations within the memory to be completed before proceeding) and an Instruction Synchronization Barrier (no other instructions are pre-fetched and the processor's pipeline is flushed). The *MSR* instruction (Move to system co-processor register from ARM register) is used to reset the *PMCCNTR_EL0* register which holds the value of the Cycle Counter, *CCNT*. Between operations we read this register with the MRS instruction (system co-processor register to ARM register). We will use this information to calculate the number of cycles (and the time) required for each operation. In figure 3 we showed the steps just described.

C. Permission Handler

The first time the *LDR* operation is performed, the hypervisor steps in to handle the exception. Since Bao does not foresee the possibility that a page can be mapped without read or write permissions, a handler is not implemented in the code to handle our case. We patched the code so that when the syndrome corresponding to the exception is encountered, a handler wrote by us is executed, instead of the error message saying that the hypervisor has no way to handle the exception.

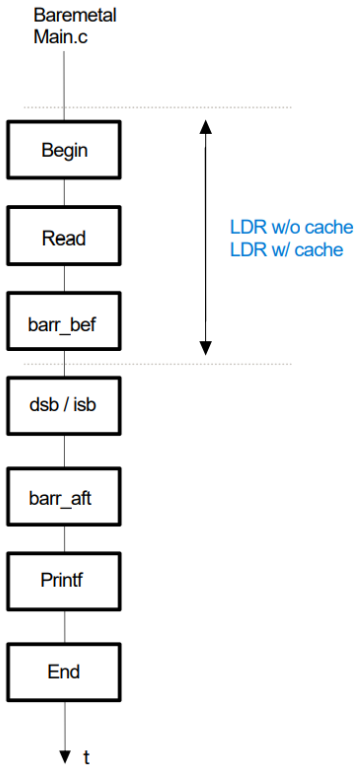

Fig. 3. The reading process in the Baremetal main.c

```
void permission_handler(uint32_t iss,
uint64_t far, uint64_t il){

uint64_t beg_handler=MRS(PMCCNTR_EL0);

int paddr = (int)far - (int)0x14000 +
 (int)0x40019000;
pte_t* pte = NULL;
pte=pt_get_pte(&cpu.vcpu->vm->as.pt,
 0x2, (void*)far);

pte_set(pte, (uint64_t)paddr,
 pt_pte_type(&cpu.vcpu->vm->as.pt,0x2),
 PTE_VM_FLAGS);

uint64_t end_handler=MRS(PMCCNTR_EL0);
}
```

The handler is quite simple. In input it receives a 32-bit integer **ISS** (Instruction Specific Syndrome), a 64-bit integer **FAR** which holds the faulting address and a 64-bit integer **IL** which holds the Instruction Length for the current exception. First we calculate the physical address (*paddr*) of the page starting from the FAR using

a base and an offset; then we get the pointer to the page table entry we are trying to modify using the function *pt_get_pte*, as input values we use the address of the page table, the level of the multi-level translation and the faulting address. After getting a pointer to the PTE, we can give all the access permissions to the page with the function *pte_set* that, as explained in the section IV-A, makes an OR between the physical address, the page type and the flags.

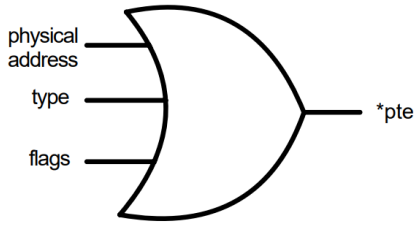

Fig. 4. The pte_set operation

After using the *pte_set* function, we could add an LDR operation to bring the page into the cache and speed up future memory reads. We want to do this because the page will have all the necessary permissions, which means it can be read directly from the cache instead of having to go to memory first. This will make the process faster and more efficient. However, when we tried to test this read operation, we were unsuccessful and didn't have enough time to investigate the problem. We think that caching the page in our handler managed by the hypervisor caused some memory misalignment, and during execution, the CPU encountered an error and stopped.

V. Results

Here below are the results of the tests we carried out on both Board and Qemu. We measured the number of clock cycles needed to perform:

- VM exit: the time from the attempted reading (that caused the exception) to the starting of the permission handler, in which the Hypervisor takes control of the VM and performs any necessary operations, such as saving the state of the guest operating system.
- Handler: the time needed to execute the permission handler (computation of the physical address, finding of the PTE address and setting of the access permissions).
- VM entry: time needed from the end of the handler to return to normal execution of the VM. The hypervisor restores the state of the guest operating system to the point at which it left off, and the guest operating system resumes execution.

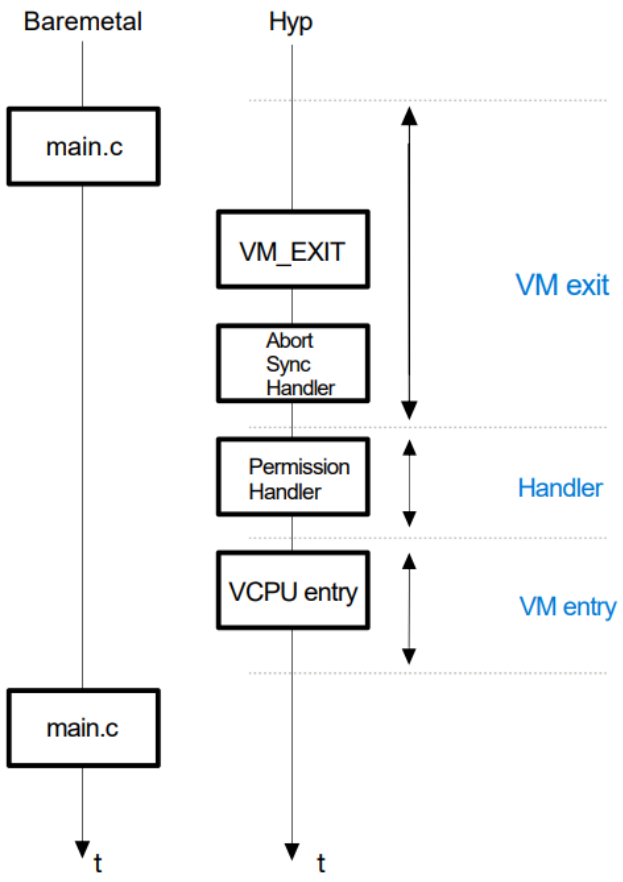

Fig. 5. workflow permission handler

In figure 5 we illustrated on a temporal line the names (in **black**) of the functions executed by the VM and Hypervisor and reported and the times we measured (in blue). We compared these times with a normal LDR operation both from a cached data (LDR w/ cache) and a LDR operation without a cached data (LDR w/o cache).

Each measurement was taken 50 times, the average result is reported in the tables in the next sections.

A. Qemu

The tests have been carried out on an emulation of the same processor of the Zcu 104 board, the Cortex-A53 with a clock rate of 62,5Mhz. Unfortunately Qemu can't emulate the cache properly and we weren't able to measure the time of an LDR operation without the cached data as the number of clock cycles was the same both before and after a cache flush.

TABLE II: EXPERIMENT RESULTS ON QEMU

Operation	Cycles	Time (s)
LDR	2.74×10^2	4.38×10^{-6}
VM exit	6.85×10^3	1.09×10^{-4}
Handler	1.47×10^3	2.36×10^{-5}
VM entry	5.60×10^3	8.96×10^{-5}

B. Board

The tests have been carried out on the Zcu 104 board, with a clock rate of 100Mhz. The times for the Handler execution and for the VM entry are much higher compared to the LDR and VM exit operations. We suppose it could be due to our data collection method, which we could tune for measurements on Qemu but could not be applied on board due to lack of time. Since the LDR and VM exit operation have a similar duration, both on Qemu and on board, we can suppose that the VM entry and Handler on board have a duration of the same order of magnitude of the first operations.

TABLE III: EXPERIMENT RESULTS ON BOARD

Operation	Cycles	Time (s)
LDR w/o cache	1.52×10^2	1.52×10^{-6}
LDR w/ cache	7.11	7.11×10^{-8}
VM exit	1.95×10^2	1.95×10^{-6}
Handler	3.98×10^6	3.98×10^{-2}
VM entry	5.84×10^6	5.85×10^{-2}

VI. CONCLUSIONS

On Qemu, we were able to successfully implement our patch and thereby restore permissions to the page, demonstrating the practical feasibility of initializing pages with Bao without permissions and returning them via an exception handler implementing in this way a fast recovery mechanism for page access restoration in Bao. We attempted to apply the same reasoning to the board, but since the source code differs, permissions are indeed restored to the page (as evidenced by the fact that the PTE in the translation table is different), but whenever we attempt to read it again, the exception always triggers. We hypothesized that this may be due to a caching or buffering issue with data storage.

In table II we can see that the times needed for the VM exit, VM entry and the execution of the handler are only an order of magnitude greater than the time needed for a LDR operation. We can assume, therefore, that the implemented permission recovery mechanism can be a valid option for fault recovery in Bao hypervisor. From table III we can see how the time needed for the VM exit operation is even closer to the time need for the LDR operation without a cached data, with this time and with the considerations reported in section V-B, we can assume that the entire permission restore operation (including the VM exit and the VM entry) lasts approximately three times the LDR operation.

While our patch showed to have a relatively low latency and overhead, further research is needed to implement the page pre-fetching mechanism that could speed up subsequent accesses to pages whose permissions have been restored, resulting in a faster recovery and reducing the total downtime of the system.

We are confident that this mechanism (with further refinements) can be implemented in other hypervisors making them more robust to such faults, thus reducing the downtime of the whole system.

REFERENCES

[1] T. Shimada, T. Yashiro, N. Koshizuka, and K. Sakamura, "A real-time hypervisor for embedded systems with hardware virtualization support," in *2015 TRON Symposium (TRONSHOW)*, 2015, pp. 1–7.

[2] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A Lightweight Static Partitioning Hypervisor for Modern Multi-Core Embedded Systems," in *Workshop on Next Generation Real-Time Embedded Systems (NG-RES 2020)*, ser. OpenAccess Series in Informatics (OASIcs), pp. 3:1–3:14.

[3] ARM Developer. 2-stage translation. [Online; accessed February-2023]. [Online]. Available: developer.arm.com/documentation/102142/0100/Stage-2translation

[4] ——. The memory management unit (mmu). [Online; accessed February-2023]. [Online]. Available: https://developer.arm.com/documentation/101811/0102/The-Memory-Management-Unit–MMU-