Concurrent Checkpointing for Embedded Real-Time Systems

by

Michael Prinke

A Thesis Presented in Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2018 by the
Graduate Supervisory Committee:

Yann-Hang Lee, Chair
Aviral Shrivastava
Ming Zhao

ARIZONA STATE UNIVERSITY

December 2018

ABSTRACT

The Internet of Things ecosystem has spawned a wide variety of embedded real-time systems that complicate the identification and resolution of bugs in software. The methods of concurrent checkpoint provide a means to monitor the application state with the ability to replay the execution on like hardware and software, without holding off and delaying the execution of application threads. In this thesis, it is accomplished by monitoring physical memory of the application using a soft-dirty page tracker and measuring the various types of overhead when employing concurrent checkpointing. The solution presented is an advancement of the Checkpoint and Replay In Userspace (CRIU) thereby eliminating the large stalls and parasitic operation for each successive checkpoint. Impact and performance is measured using the Parsec 3.0 Benchmark suite and 4.11.12-rt16+ Linux kernel on a MinnowBoard Turbot Quad-Core board.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

Chapter 1

INTRODUCTION

1.1 General Description

Embedded applications are rapidly increasing in complexity while maintaining a requirement of real-time performance. This stems from growth of the Internet of Things (IoT) ecosystem and the hardware supporting it. Most platforms within this space do not support hard real-time as they are derivatives of consumer products such as control gadgets in unattended environments. The debug capabilities of these platforms is also more complicated since they tend to be headless without a system level user interface and when deployed cannot be taken offline for debug activities. This deduces debug capabilities to events that can be easily reproduced on similar hardware that may not be completely identical. Error scenarios are exponentially difficult to reproduce when long hour testing is required. Even more so when a best known method of steps to reproduce the error are not available. These scenarios are best addressed by the generation of application state checkpoints which can be replayed on similar hardware with replicated or simulated operations from IO devices.

Generation of a checkpoint within a real-time system requires intricate understanding of system behavior, resource capacity, and resource bandwidth. Within a hard real-time context, much of this information is known at design time. This is possible in hard real-time targeted systems due to bounds on the execution behavior as well as additional hardware capabilities to enforce maximum latency and constrained resource consumption. In a soft real-time context, this information may be known within some statistical limits and only limited capabilities for reducing latency maximums. For instance, Time Division Multiplex-

ing (TDM) is common for hard real-time systems. However in soft real-time, priority-based scheduling and resource isolation is more common to avoid high latency spikes while context switching, interrupts, other asynchronous events are still allowed. When adding checkpoint capabilities to a system, there is a high risk of impacting the cycle time or response time of real-time applications. The main concerns are multiple consumers of the data, a dependency that now exists in synchronizing the checkpoint across multiple application threads, and the sharing of hardware resources.

This work aims to perform a majority of the checkpointing effort concurrently by tracking dirty pages and copying them asynchronously, there by reducing the checkpoint latency observed by the target application. This starts by tracking the application state, compiling all state changes into a synchronized checkpoint, and saving off the checkpoint to a storage medium. The goal is to limit the total observable overhead to the application with checkpointing enabled. Capabilities can compliment various methods of bandwidth management and checkpoint scheduling. With the interest of improving the ecosystem, the solution with be a software only advancement with reliance on current hardware capabilities found in the Intel Atom Processor family. While this work is targeted at embedded real-time systems, it can be applied to enterprise applications.

1.2 Significance of the Problem

As the software industry evolves for embedded applications, multiple parties will be involved in the development and maintenance of IoT systems such that software is becoming less likely to be designed in-house, and multiple software vendors may play a role in any given design. Support of these systems requires a more robust debug tool-chain with concurrent checkpoint and replay at the forefront. This will reduce the debug hours spent on

potential errors, and the impact observed by the end customer of the product. By regularly checkpointing an application, the most recent checkpoint prior to an error occurrence can be provided to a debug team. The debug team can then use the individual checkpoints to replay and identify the error to develop a fix. When using in-place patching of software, this method can provide a method of verification and regression to prove a bug is fixed. A collection of checkpoints can also provide the debug team visibility into a possible use case that was not identified and addressed at design time. Without checkpointing, an identical setup is typically required that can be time consuming and expensive to house, maintain, and manipulate for debug practices. By studying the impact of concurrent checkpointing on embedded applications, the overhead can be better understood as it relates to varying types of environments and workloads.

1.3 Problem Statement and Scope

Checkpoint and replay of applications has been a classical approach for fault tolerance and process migration. For long-running embedded applications, it is becoming common practice for debug. The replay of errors and faults is made possible by restarting an application from a known state saved off by a successful checkpoint solution. In a real-time embedded system utilizing commodity hardware, software based monitoring must be used in place of a hardware based monitoring solution that may not be readily supported. A software based solution also gives way to broader adoption and evolution much like the operating system and system libraries. The goal of this work is to research and propose such a solution for concurrent checkpointing within the constraints of a multi-core, multi-threaded, embedded environment where resource bandwidth limitations are observable and measurable by the software developer. This includes defining an initial state of an application prior to check-

pointing, the methods used to track changes in application state such as virtual and physical memory, and the cpu state at a point in time for the checkpoint to take place. Most importantly, the checkpointing mechanism targeted in the research must carry out a majority of its operations concurrently during the execution of applications threads. Hence, there should be minimal interference to the application's execution nor should the logical behavior of the application be altered.

The scope of the research is limited to the various techniques for concurrent checkpointing including the tracking of state changes, copying the changed state, and replaying from that state. This excludes a natural improvement that would address monitoring of IO devices for simulating IO responses during replay from a checkpoint.

1.4 Thesis Statement

Using an Intel Atom processor running Preempt-RT linux and a multi-threaded soft real-time application, the application state changes can be tracked using software and saved off in a scheduled fashion to enable concurrent checkpointing for future replay. The checkpointing mechanism needs to have a limited and quantifiable impact to real-time performance and the response time of embedded application to be checkpointed.

1.4.1 Objective

To improve the use of checkpoint methods using a commodity grade solution with little to no proprietary intellectual property. The setup should closely resembles a simple soft real-time system, rather than one that meets the requirements of a hard real-time. Real-time performance is not widely represented by benchmarks within the open source community

and at the time of research benchmarking suites requiring licenses such as EEMBC were not available for use. The re-purposing of an existing benchmark, PARSEC 3.0, should be possible to meet the demand of a real-time application and provide demonstration of both overall performance and response time. The application life cycle should be composed of an initialization phase where memory is allocated and resources are defined prior to a *Region of Interest* where the application performs an operation repetitively. During this repetition, checkpoints of the application state should occur concurrently with little to no added overhead to the application. The application state changes are collected and saved off to non-volatile storage medium to be used for future replay of the application.

1.4.2 Procedure

The hardware selection is a MinnowBoard Turbot Quad-Core board aimed at the maker community and Internet of Things ecosystem. The operating system is Ubuntu 18.04 Minimal Install combined with a modified 4.11.12 linux kernel and the 16+ Preempt-RT patches applied. The linux kernel is configured to isolate 3 of the CPU cores from system interrupts and scheduling. The PARSEC 3.0[1] benchmark is employed as the testing means with instrumentation to measure response time of each compute operation and the added synchronization between dissimilar threads. After initialization is complete, but prior to the *Region of Interest*, the application is stalled while an initial checkpoint is created using [6] to assist in restoring both application and system state during the replay phase. Once resumed, the application enters the *Region of Interest* and the concurrent checkpoint methods begin.

Chapter 2

LITERATURE REVIEW

2.1 Relevant Theory

The basis for concurrent checkpoint spans many areas of work with the two ends of the spectrum being migration of virtual machines with little to no concept of real-time and managing functional safety under hard-real time constraints. The migration of virtual machines evolved quickly with the rise of the data center relying mostly on software based solutions due to the limited hardware support provided by mainstream products at the time. Hypervisors began to employ solutions to reduce the downtime of migration by copying data prior to a checkpoint and using dirty taking bits in the page table to monitor which pages during checkpoint must still be copied. This capability of dirty tracking is supported by Intel Architectures where a hyper-visor is present. Consequently this hardware advancement does not work in native more where vitalization is not deployed. Containers begin to take off where virtual machines leave off since in the case of containers to have the "state" which must be saved, copied, and restored and unlike virtual machines, no hyper visor is present since the container will run native to the operating system providing the service. Focusing more in the application domain, there are two prominent solutions being CRIU (Checkpoint and Replay in Userspace)[6] and BLCR (Berkely Lab Checkpoint and Replay)[8]. BLCR is loosely maintained within the 3.X linux kernel, due to heavy kernel modifications in order to operate. CRIU being less kernel dependent is current and stable with modern Linux releases and make use of ptrace capabilities for OS State Management. Both solutions provide a means to halt and application, save it's state to a storage medium, and restore at a later time on

like hardware and software. CRIU is particularly useful for debugging complex applications such as the Firefox web browser since it also supports common OS state issues like multiple threads, file pointers, and inter-process socket communication. BLCR is more limited in its support and therefore was not tested along side CRIU for this body of work. Thus far, the checkpoint methods mentioned have focused on common desktop applications with little interest in real-time systems or embedded systems. In the real-time embedded space, most research has been dominated by hard-real time use cases where deterministic monitoring is employed. In-order to meet the deterministic nature of the monitoring, all bodies of work mentioned are hardware based solutions typically monitoring the memory traffic either inside the CPU core, or on an external coherent memory bus. These solutions, rather than monitoring which segments of memory are changed, seek to track the coherent memory state on a per transaction basis. The replay in these systems also trend towards a deterministic solution that can be used to satisfy functional safety needs, thereby running two applications in parallel, and if one fails or is incorrect, the next frame it can restart from the successful applications known state. The interest of this work is to evaluate concurrent checkpointing methods for soft real-time embdded systems, and therefore revolves around advancing the CRIU solution with advancement when possible from other relevant areas.

2.1.0.1 Hardware Based Monitoring

Hardware based monitoring consist of techniques that require no additional sets of instructions inline with execution code. Many techniques reviewed utilize hardware features for logging that are saved in buffers or registers that must be moved to memory or non-volatile storage which add additional overhead either through additional out of cycle instructions or bandwidth requirements on the memory bus. In [20, 21], Tsai discusses a hardware

monitoring solution utilizing the address and data signaling commonly founding in older microprocessor designs when the memory hierarchy was separated from the CPU itself. Although in this case a full monitoring solution is provided, a significant amount of data is sampled on each clock cycle and then used to enable replay and re-execution at a later time. The value in this research provides targeted logging specific to events and function calls to reduce the overall logging resource requirements. Implementing this such solution on today's modern architectures is inherently very difficult due to the speed of executions and resulting data to capture at this level. Intel provides Last Branch Record tracing capabilities which allow a user to track basic blocks in clock tick granularity[intelsdm3b]. This provides tracking for individual threads but with limited buffering since latest architectures only support 32 entries. Also since the trace entries store clock ticks which are based off a variable frequency, cross correlation is not possible with neighboring cores on the same die. Work can be done to enable correlation and use a means to track execution flow but at 32 entries, the buffers must be record extensively with added check pointing of data in heap and stack. DeLorean[13] provides a chunk based hardware solution to monitoring with a companion replay mechanism. DeLorean is the most advances solution thus far when using a hardware based solution but falls short due to the tight coupling a cpu's execution units. It also is affected by additional stalls due to how micro-architectural issues are handled including interrupts, branch perdition, and cache overflow. BugNet[14] also integrates with the execution units and uses simple checkpoints including minimal information of the architectural state such as program counter and register file contents. Unlike full check pointing providing a known state of memory, all first time load memory accesses are included in the monitoring, removing the need to checkpoint memory. the key limitation for embedded systems is the hard limit on the replay window the architecture has.

2.1.0.2 Software Based Monitoring

Software based monitoring is defined as the addition of instructions that must be executed inline that add additional delay to the total execution time. Intel has introduced the System Visible Event Nexus (SVEN)[19] in its recent product offering which provides low interference in the range of 20ns[2]. SVEN uses binary storage of 32 or 64 bytes which includes a time-stamp and short header. Due to the low cost of SVEN it makes it the ideal candidate of monitoring real-time workloads withing a multi-core environment. Since SVEN is an on-die solution it does not directly provide correlation of events in a distributed environment, but can be extended with logical clocks to do do. ARM provides a similar capability that adheres to the MIPI® System Trace specification[5]. This allows an application developer to place trace events inline with execution code as a low cost solution to print statements. The claim of low latency is made but with no supporting data or comparison.

2.1.1 Check Pointing

Li[11] compares two check pointing solutions that satisfy real-time environments by copying memory contents concurrently. This works by using a concurrent copy thread to read contents in memory and mark the virtual pages as read only. Therefor, if any write occurs by the executing thread, a page fault occurs. The small memory system concurrently writes the checkpoint out to a slower medium but forces extended delays if a page of memory is accessed while waiting to be copied. In a similar scenario with large memory availability, the copying can occur much quicker, but the inherent problem of page faults still exist. The problem in modern systems now is that the TLB structures prevent painful page translation misses that can have large latency. Additionally since the virtual pages are being updated at

run time, this flow is common practice for self modifying code, which requires flushing of the TLB structures that further add latency to execution. While Li's research aims to target real-time, the allowable overhead is 100ms which is much grater that most real-time responsiveness requirements seen today. In [15], a checkpointing solution is proposed that uses synchronized clocks rather than logical clocks such as lamport clocks and seems to ignore the checkpointing latency itself due to the nature of the distributed system. In [3] and [23], the algorithmic cost of checkpointing is managed through an adaptive mechanism understanding both time and energy restrictions on the system. Zhang's algorithm for adaptive check pointing targets fault tolerance where faults arrive as a Poisson process with rate λ. This algorithm assumes a rollback on fault and repeats the execution still within the deadline. Since the goal of replay is debug rather than fault tolerance, the correlation of faults could promote a desired window size to detect faults and use the information to eliminate soft errors from design related faults. A GDB based solution also called Delorean[12], not synonymous with [13] as a hardware monitoring solution, provides checkpoint and rollback capabilities. the methods used to reduce memory usage for checkpointing and the most state-of-the-art. Since DeLorean depends on high speed memory backed storage rather than the slower non-volitile storage mediums, the usability is deminished for a closed chassis hard real-time system. RR[17, 16] is a comparable solution aimed at debugging the Firefox web browser, but does not provide a checkpointing and replay mechanism safe for live execution for replay at a later time on a remote system. Being open source, rr provides a sandbox for early research, but does not port well to deep embedded RTOS solutions. In [4] a checkpoint solution for containers is proposed based on flagging of dirty pages and copying the data concurrently with execution until a checkpoint barrier occurs from which a checkpoint must be finalized. The algorithm seeks to control the time spent during pre-copy versus stop-and-copy stages

to reduce overall time spent on checkpointing. [22] creates a compile in software checkpointing that creates a shadow copy of memory during program execution.

2.1.2 Other Related Work

DMP[7] seeks to enforce deterministic shared memory in known non-deterministic multiprocessor environments. This can provide an optimized means to create locations for check pointing since it is another way to drive synchronization without traditional synchronization barriers. The value is in the quantum definition of where to place the token passing. Noticing that the hardware solutions provide speed up where the software solution provides considerable slow down, it is not likely a viable software solution for real-time systems.

Chapter 3

METHODOLOGY

3.1 Design

3.1.1 Dirty Page Tracking and Concurrent Checkpoint Process

The basis of the design utilizes the hardware mechanism of page faults to track memory state changes. This provides support for multi-thread and multi-core environments. The page faults occurs when physical memory is either not mapped to a virtual memory address known as an *anonymous page*, or does not have the correct permissions, such as writing to a read-only page knwon as a *dirty page*. The soft dirty tracking defines the use of marking writable pages as read-only, and where a write permissions fault occurs, the page is granted write permission and marked as dirty by setting a bit in their respective page table entry. To avoid parsing the entire page table to discover dirty pages, and dirty queue is implemented which is populated at the same time the bit in the page table entry is set. The process performing the checkpoint can then retrieve the dirty pages asynchronously from the dirty queue. This tracking comes with an overhead which is discussed in the Chapter 4. Since the memory state changes must be synchronized to a known valid state for replay, all writes to memory after a checkpoint boundary must be prevented from modifying the physical memory which may yet to be collected by the checkpointing process. Therefor the ckpt_inprogress flag is used to signal the beginning of a checkpoint boundary. When a page fault occurs and the flag is set, then the page fault will wait for an event signaling the checkpoint process is complete. Once the ckpt_inprogress flag is set, the checkpointing facility should make sure

all pages are marked as read only such that any write to memory may be captured, and forced to wait for the checkpoint finished signal event.

In order to properly checkpoint a multi-threaded application where multi-core scheduling is possible, each thread must be checked in the kernel runqueue if it is scheduled and running. If it is not running the cpu state can be saved from the scheduler task struct. If it is running, then an smp_function_call is forced on the core where the thread is running, forcing it to swap, and save the cpu state to the scheduler task struct. From the task_struct the cpu register state can then be retrieved and saved for checkpoint replay. In the case of more threads than cores, the runqueue can be modified to prevent other threads from running in place before their cpu register state can be collected.

With all thread's cpu register state collected, the dirty page queue must be emptied and all physical memory changes remaining must be delivered to the checkpointing process in user space. In the event of Virtual Memory Map changes, the entire VMA table is saved off during each checkpoint and delivered to user space after the cpu register state is collected but prior to the dirty queue being emptied the last time. Once all state changes are made available to the checkpointing process in user space, the application selected for checkpoint can continue. The Checkpointing process in userspace is responsible for saving of the final checkpoint state to persistent storage. This flow with respect to dirty tracking is presented in Figure 3.1.1

3.1.2 High level Design - Checkpoint

The checkpoint infrastructure consists of kernel modifications to existing page fault flows and an accompanying kernel space driver called the creplay driver. The intelligence for algorithm selection and orchestration occurs separately in a user space application, the

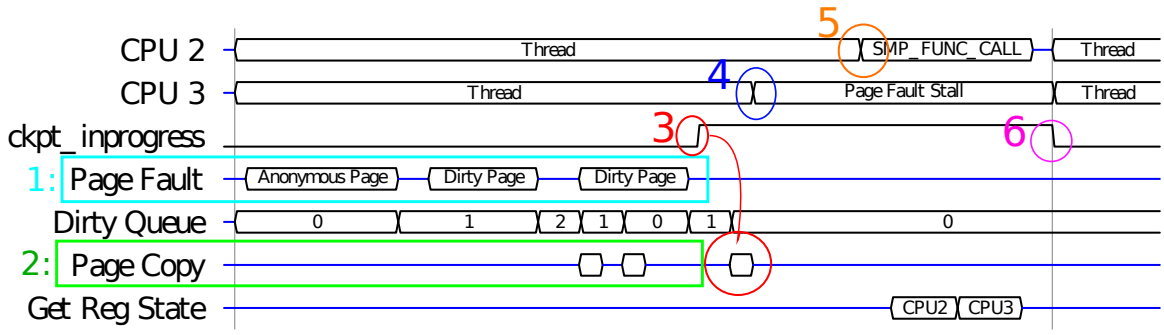


Figure 1. Waveform Representation of Concurrent Checkpoint Activity with 2 Thread
1:The dirty queue fill by page faults for both anonymous pages and dirty pages.
2:The dirty queue is access asynchronously by the checkpointing process and pops one item from the dirty queue at a time, copying the contents of the physical memory for that page and marking it once more as read-only.
3:After some time, the checkpointing process will request a checkpoint and set the ckpt_inprogress flag and immediately pop all remaining items from the dirty queue and marking all pages as read-only.
4:Thread on CPU 3 has attempts to write to memory while ckpt_inprogress is set, a Page Fault Stall Event occurs where the thread waits for the ckpt_finished_event.
5:The checkpointing process then requests the cpu register state of all threads within the process. With Thread on CPU 2 still running, an smp_func_call is sent to that core to collect the thread's cpu register state.
6:ckpt_inprogress flag is unset creating a ckpt_finished_event allowing the application to continue normally.

checkpointing process, where the checkpoint data is collected and saved to persistent storage. The application selected for concurrent checkpoint has no knowledge of or interaction with the underlying kernel hooks. Monitoring of the physical memory changes occurs through the dirty page tracking. The kernel driver provides a facility that is flexible to multiple use cases of such that the user can control the frequency of checkpointing and frequency of concurrent page copies. It is extendable to support a per page designation to reduce page faults based on categorization such as type (ie. stack memory), or frequently written. Therefore the checkpointing facility provided by the kernel driver can be reused by a different user space application tuned to fit the needs of the target system.

3.1.2.1 Workload Capabilities

There are no required changes to support concurrent checkpointing of an application. Any application can be launched by the operating system, and intercepted by the checkpointing infrastructure. For applications where a known initialization phase occurs and threads are spawned, a checkpoint syscall has been added that allows an application to stall and wait until the checkpointing can begin. This sis call is defined as sys_checkpoint() with syscall number 333. The PARSEC benchmarks used for testing have this syscall added at the beginning of the *Region of Interest* to properly intercept and measure the overhead of checkpointing.

3.1.3 Detailed Design

3.1.3.1 Concurrent Checkpoint Manager

The userspace application for controlling concurrent checkpointing is a C++ application that utilizes the Checkpoint and Replay Kernel Interface provided by /dev/creplay. The application is run from the command line using the arguments from Table 1. Only the main thread context is used and shared with all kernel level operations. At initialization time, two RB-Trees are created to store the PTEs and VMAs representing the virtual and physical memory of the application to be checkpointed. PTrace is used to halt execution and stall the threads initially to collect a baseline, but is not used in the concurrent checkpoint flow. A pre-checkpoint is requested of the kernel[1] as a baseline for all future checkpoints. If the

[1]Described by Section 3.1.3.3

threads are stalled in the checkpoint syscall when the pre-checkpoint is requests, they will begin executing after PTrace detaches.

Once all threads are running, the concurrent checkpointing begins by calling the POP_QUEUE ioctl command from the kernel driver to collect pages in the dirty queue. All data from the POP_QUEUE commands is persistent in memory until processed at the end of the checkpoint. The POP_QUEUE repeats instantly if the dirty queue is not empty. Once empty, a pause of 1/10th of the checkpoint period occurs and then the POP_QUEUE is repeated again followed by another pause. This continues until the end of the checkpoint period is reached. To initiate the checkpoint, the POP_QUEUE command is called once more with the ckpt_inprogress flag set. This represents the beginning of the checkpoint boundary. The POP_QUEUE command repeats until the dirty queue is empty once more. All pages within the target application should be marked read-only at this point. If any pages in the application have dirty tracking disabled for performance reasons, these pages should temporarily be marked as read-only. The VMA data is then dumped next using the VMA_DUMP ioctl command. The GET_REGS ioctl command is then called next to collect the CPU register state of each thread to be checkpointed. Finally the POP_QUEUE command is called once more with the ckpt_inprogress flag unset to signaling that the checkpoint is complete, and the threads can continue. All the data must be processed at this point. The PTE data from POP_QUEUE commands is queried against the data in the PTE RB-Tree using the virtual address of each page. If the address does not exist in the tree, it is added and marked as "new". If it does exist, and any memory changes are found, it is marked as "dirty" with the new contents copied into the existing data. Similarly with the VMA_DUMP data, the VMA data is queried and flagged for changes. If the VMA region is new, it is added. If it has shrunk, moved, or expanded, it is marks appropriately. All changes to the physical pages are stored then written to a *.pte file, all vma changes are written to a *.vma file, and

the CPU register states for each thread are written to a *.regs file. Finally the event log is retrieved using the GET_LOG ioctl command and the checkpoint counter is incremented. The concurrent checkpointing begins again by the repeated calls of POP_QUEUE until the checkpoint period expires once more.

The copying of physical memory from the application is done by the kernel driver when the POP_QUEUE command is issued. The checkpoint process is responsible for creating a buffer adequate in size for the kernel to copy the memory contents into. The design choice to use the kernel driver to perform the copy is an optimization from the way both CRIU and PTrace supports. CRIU uses parasite called "Compel" to send the contents of memory to the CRIU address space. Since the parasite takes over the threads, this intrusion was to be avoided. PTrace also supports a means for one application to collect that memory state of another by effectively mapping the target address into the consumers address space. This was one possible solution but remained difficult in synchronizing with marking of each page as read-only to correctly maintain the flow for correct dirty tracking. For this reason, the copying of the memory contents is absolutely necessary when popping from the dirty queue and using a separate facility such as PTrace versus performing the copy in the kernel driver, did not seem like a natural design choice. When the memory contents are compared against the data in the RB-Tree, any dirty pages would result in an additional copy in memory, and then copied once more by the operation of saving the checkpoint state to persistent memeory. The overhead of these two final copies are not observed by the target process, therefor no attempts to avoid them are made.

Argument	Description	Default
-m <mode>	Mode of operation. Options: pid, concurrent,concurrent_overhead concurrent_verify, squash, squash_verify, replay_file	N/A
-o <filepath>	Basefile name for output of checkpoint files. Creates “filepath”[.pte,.regs,.vma]	N/A
-c <filepath>	Checkpoint file to use for replay	N/A
-p <pid>	PID of main process of focus for checkpoint/replay	N/A
-s <size>	Maximum size of page faults allowable per checkpoint. Size up for larger checkpoints	4096
-b <begin>	Checkpoint start number for squashing checkpoints	0
-e <end>	checkpoint end number for squashing checkpoints	0
-f <freq>	Target frequency for checkpoint operations	100

Table 1. Execution times of Blackscholes benchmark program

3.1.3.2 Kernel Modifications

All Kernel Changes are wrapped and enabled by the CONFIG_CREPLAY define at compile time by the preprocessor. The KConfig additions are listed in Table 2 similarly with a config file used during compilation located in the source tree as rt-creplay-config which can be renamed to .config to reproduce the setup. The mm_struct structure is used as the primary construct for storing checkpointing data. This includes the dirty queue defined as struct pte_checkpoint_queue, the queue for the event log, a spinlock for checkpoint synchronization, and multiple flags for managing state including the ckpt_inprogress and ckpt_enabled flags. The vma_struct has the added unsigned char * pte_cpt_stage which stores the state of each page within that vma context. The task_struct has a added flag for logging when the thread is stalled in a page fault event.

The Soft-Dirty tracking used by CRIU is reused with slight modifications to track the events and add the pages to the dirty queue. The page fault flow is first modified in the

Config Name	Description	Default
CREPLAY	Enable/Disable Support of CREPLAY Driver and associated Kernel Hooks	Y
CREPLAY_QUEUE_INIT_SIZE	The default queue size for saving off which dirty pages must be copied.	1024
CREPLAY_NUM_INSTANCES	Number of allowable concurrent checkpoint instances	8

Table 2. CREAPLY KConfig Settings

early flow within the handle_mm_fault() function. Here is where a forced stall occurs if the thread's ckpt_inprogress is set. If it is, it enters a wait_queue to be later woken up with the checkpoint is complete. The do_anonymous_page() and handle_pte_fault() functions are where the page tracking occurs and additions to the dirty queue are made. Within handle_pte_fault() the pte_cpt_stage is updated to reflect the current state of the page in the checkpointing process and logs if requeueing has occured[2]. The ckpt_event() logging function is added to assist in tracking all concurrent checkpointing events in the kernel and eliminate the need for printk style logging. Logging is also added to the context_switch() function to log context switch events[3]. The associative relationship for data structures within the checkpoint facility is shown in Figure 3 with arrows representing pointers and dots representing instances. The checkpoint_file exists as a static array within the creplay kernel driver supporting 8 instances by default.

3.1.3.3 Kernel Driver Interface

The kernel driver "creplay" houses the bulk of the changes including a syscall which is why it is a compile-in driver rather than a module. The purpose of the creplay kernel

[2]The requeue counter is not used for any purpose at this time.

[3]the logging in the context_switch() function does not log all possible swap events as some events may be missing from the final log

driver is to avoid halting the threads to retrieve the application memory state and track activity in a low cost manner. The IOCTL commands are listed in Table 3. The typical flow will follow the state diagram in Figure 2. The first step in any concurrent checkpointing scenario is the checkpoint syscall which will stall the threads until woken by the creplay driver. A syscall in this case is used instead of ioctl to simplify integration with an application of any permission level. The concurrent checkpoint infrastructure starts with issuing a CHECKPOINT_MSG or CHECKPOINT_STALL_MSG. This saves of a baseline of the application state, initializes the soft-dirty tracking, and wakes the threads(s) from the checkpoint syscall. At this point the application is being actively monitored, and Dirty Pages are available by the POP_QUEUE ioctl command. If POP_QUEUE is issues with (1) argument, representing ckpt_inprogress, then the application's environment will transition state and any future writes to memory should be prevented, but execution will not stop otherwise. VMA_DUMP_MSG is should be used in this state to prevent added latency once all threads are halted or stalled. The GET_REGS ioctl changes the application environment state by forcing all thread to halt using an inter-processor interrupt (IPI) through the smp_func_call mechanism. The application is then returned to the running state by issueing POP_QUEUE with ckpt_inprogress cleared.
Concurrent checkpointing terminates with the thread, there is no mechanism to turn off concurrent checkpointing at this time.

Concurrent checkpointing requires multiple asynchronous activity, including in areas where preemption is not favorable. Spinlocks are in use in various flows but only for very short moments, ideally less than 100 clocks worth of instructions. Both the Dirty Queue and Event Log are wrapped by spinlocks to support multi-threaded applications. The checkpoint also has a process wide spinlock for simplicity that is used to protect the pte_cpt_stage of each vma_struct. This could be optimized similar to or directly reuse the kernel function

pte_lockptr(). Mutex locks are use to protect the allocation into the checkpoint_file array. The is only used during the checkpoint syscall, PID_MSG ioctl, and CHECKPOINT_MSG ioctl flows.

Checkpoint initialization, ckpt_queue_init(), occurs during the checkpoint syscall, of CHECKPOINT_MSG. This allocates the pointers for the pte_cpt_stage for all the vma_sturcts currently present, and initializes the Event Log, Dirty Queue, and both Wait Queues. The wait queues are used as a synchronization method between the creplay driver, and the process being checkpointed. The ckpt_stall_event wait queue is used by the creplay driver, to wait until the process enters a valid state for initial checkpoint. Once the checkpoint syscall is made, a wake event is sent to the ckpt_stall_event. The ckpt_finished_event wait queue is used by the threads when a page fault occurs when ckpt_inprogress is set. Once the checkpoint is complete for a thread and ckpt_inprogress is cleared, a wake event is sent to ckpt_finished_event. The event log must persist after a process has terminated in order to read all events. For this reason it is a statically allocated in the checkpoint file, rather than dynamically at run time. A pointer is placed in the mm_struct of the process that it belongs to at that time. The event log tracks the events listed in Table 4. The event are tracked using the rdtscp[10] instruction to get the most accurate timing possible. Further serializing of the instructions using cpuid is not favorable and dismissed to reduce the logging overhead.

3.2 Replay Design

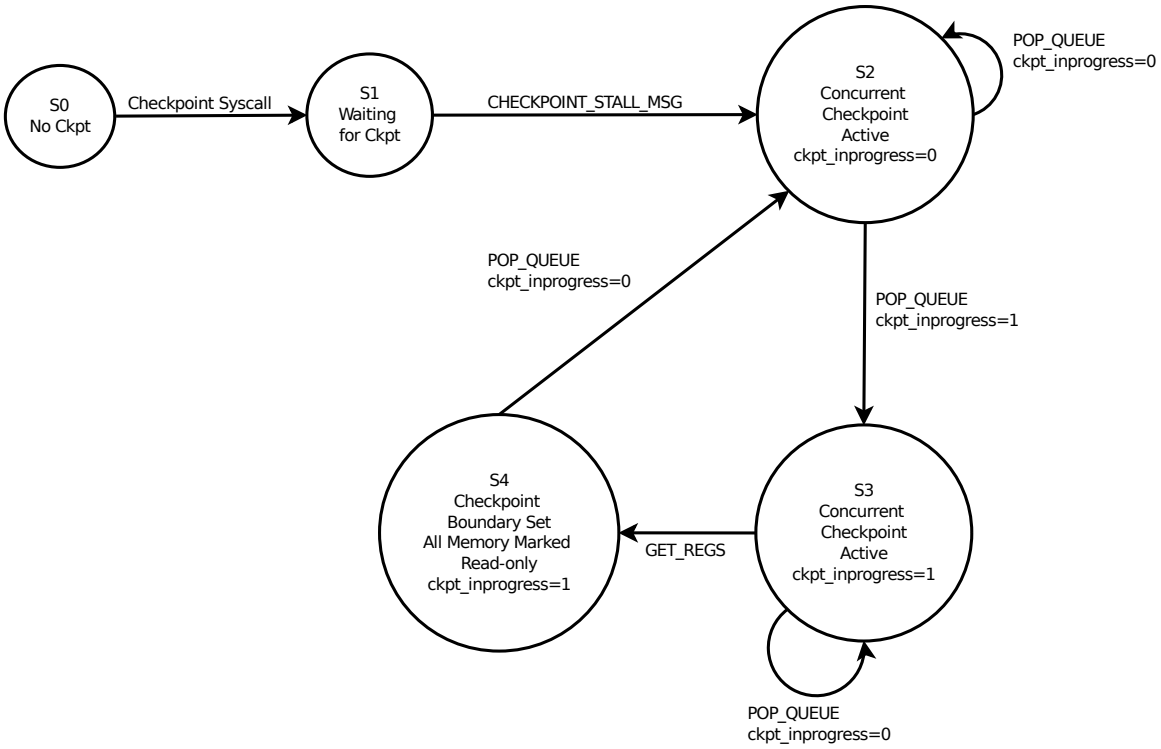
The replay infrastructure uses the same components as the checkpoint flow but with extra functionality. In order to correctly replay using the concurrent checkpoint data files, the process must have identical system level resources including PIDs, file pointers, etc. These items are not critical to the concurrent checkpoint research and therefor leverages CRIU to

IOCTL Command	Description	Data
PID_MSG	Check if PID is active in the Checkpoint Tables and Cleans up non-existant threads	pid
MONITOR_MSG	Enables Sift Dirty Tracking for a specific PID	pid
WRITE_VMA_MSG	Modifies the contents of a vma_struct, if it does not exist, create it	pid and vma_struct details
WRITE_PTE_MSG	Modifies the contents of a PTE. Must exist within a vma_struct but can map new physical memory if it does not exist	pid, 4K page contents, virtual and physical address
POP_QUEUE	Pops items off the Dirty Queue and returns the physical memory contents and virtual remapped kernel address. Sets ckpt_inprogress which defines a Checkpoint Boundary	in:pid out:virtual, physical address, and 4K page contents
GET_REGS	Collects the register State of the specific PID(s). Issues an smp_func_call if PID(s) are running	in: pid(s) out: x86 and x87 registers
VMA_DUMP_MSG	Collects the Virtual memory map for a PID	in:pid out:vma_struct(s)
READ_PTE_MSG	Reads the Physical memory of a PIDs address space. Uses translated address if present	in:pid,address out:4K data
GET_LOG	Collects the event log of Checkpoint events	event log entries
CHECKPOINT_MSG	Dumps all physical and virtual memory contents to a file and enables/clears Soft-Dirty tracking bits. Note:Assumes the Targeted PID is halted but is not required.	pid
CHECKPOINT_STALL_MSG	Waits for an application to enter a stalled state from syscall. Then same as CHECKPOINT_MSG	pid

Table 3. CREAPLY ioctl commands

checkpoint the system state prior to concurrent checkpoint, and similarly to restore the process and system state. For a majority or real-time embedded applications, this is valid since the initialization phase of applications will setup all system state, I/O, and memory prior to regular execution.

The files from the concurrent checkpoint are not immediately consumable for checkpoint since they describe changes between each checkpoint. To create a single restore point, the collection of checkpoints are squashed together along with the pre-checkpoint application state. These resulting files are then loaded into the Red-Black trees for VMA changes



State Diagram representing on CREPLAY IOCTL Kernel Interface
S0: New process, Checkpoint Uninitialized
S1: Process Stalled for Checkpoint Initialization
S2: Process Running With concurrent Checkpoint
S3: Process Marked for Checkpoint Boundary
S4: All Threads Force Stalled for Checkpoint Boundary

Figure 2. State Diagram representing on CREPLAY IOCTL Kernel Interface

and PTE changes. The PID selected for replay is then halted along with all child TID using ptrace attach. Each change noted in the red-black trees is written to their appropriate VMA and PTE using the WRITE_VMA_MSG and WRITE_PTE_MSG commands to the creplay kernel driver. CPU state is written using ptrace SET_REGS and SET_FPREGS. Finally the threads are restarted using ptrace detach.

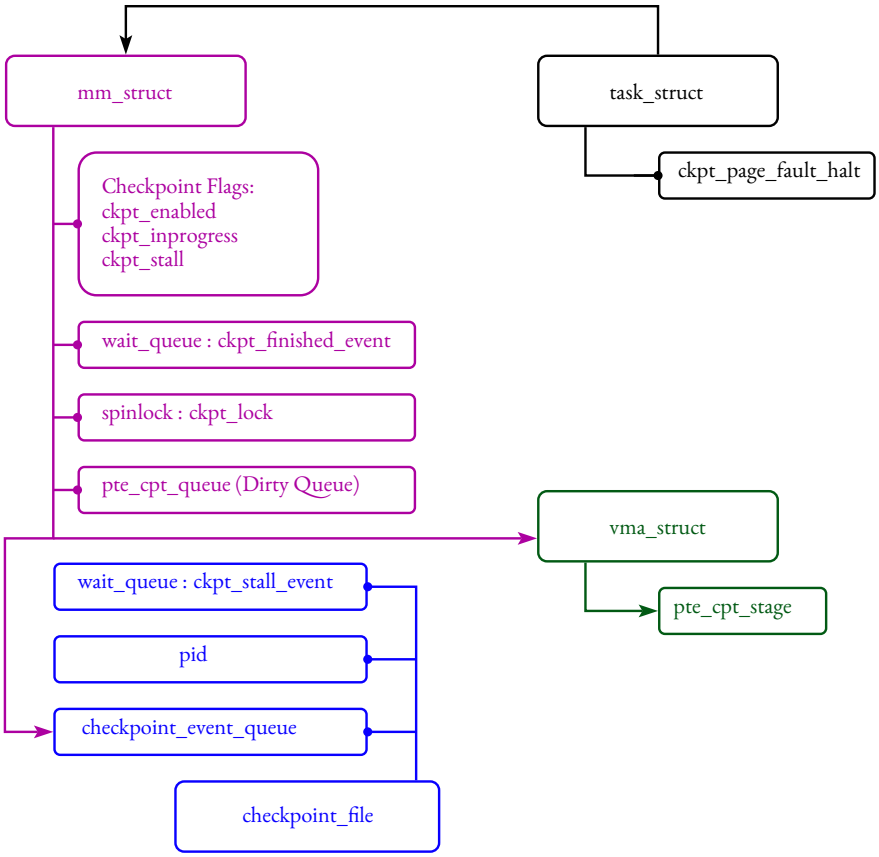


Figure 3. Relevant variables and their association to relevant kernel structures.

3.3 Additional Checkpoint, Monitoring, and Replay Techniques Not Addressed

In order to properly replay execution, certain events during execution must be repeated that otherwise wouldn't during replay from a static checkpoint. These areas are identified with a purposed solution but not included in testing due to limited scope.

3.3.1 Copy on Write

Copy on Write (COW) is theorize to further reduce the overhead of a checkpoint by preventing the application for waiting for the ckpt_finished_event when a page fault occurs

when ckpt_inprogress is set. The idea is to copy the memory contents of that page to a separate buffer that can be retrieved much like the dirty queue and allow the thread to continue running. This was not extensively tested as there is complexity in managing the memory state, and correctly collecting the cpu register state since the thread is known to be crossing the checkpoint boundary. A few of the requirements have been implemented such as the data_checkpoint_queue but a separate retrieval method is necessary that also marks the page as read-only after the ckpt_inprogress flag is unset and the contents are read in the following checkpoint period.

3.3.2 Syscall Monitoring

Syscalls can be trapped by ptrace that allows for both monitoring and manipulation. This is one simple solution to have the concurrent checkpoint thread trap on syscalls from the real-time application. The issue is the overhead of trapping syscalls which requires extra scheduling delays and can cause serialization of syscalls where they otherwise wouldn't be. One possible solution to avoid this is to manipulate the syscalls in the kernel to support concurrent monitoring, and replay when the data is available. With hundreds of syscalls, all scattered through out the kernel, this is a larger effort than what can be achieved as part of this work.

3.3.3 Syscall Replay

Replaying syscall behavior is possible using ptrace just as it is for monitoring. All syscall data can be stored to a file, and used for replay. Where replay is not required to run at real-

time speeds, added overhead is not a concern. For replay, manipulation of the kernel is less likely, although knowledge of each syscall would be necessary for correct operation.

3.3.4 RDTSC Monitoring

RDTSC and RDTSCP provide immediate access to the cpu clock without costly syscalls. Since it is natively supported by the Instruction Set Architecture and allowed in User Space there is no simple trapping on execution of the instruction. It is possible to trap with a hypervisor, which is common in real-time embedded systems, this adds additional complexity to the test setup that is out of scope. An ideal solution would trap on execution of the instruction, and save off the value to a space where the concurrent checkpoint can retrieve it at time of checkpoint without overhead to the application. A hypervisor such as JailHouse[18] or ACRN[9] is used in these environments and can be modified for this support.

3.3.5 RDTSC Replay

Replaying of RDTSC values is more difficult since the instruction still requires trapping through hypervisor. In this case for each execution window, a data buffer of RDTSC values would be passed to the hypervisor, from which all trapped calls could be linked and repeated. Instruction replacement is not possible since RDTSC has too small of a byte count.

3.3.6 MMIO Monitoring

Device or Sensor Data is typically received through a Memory Buffer commonly referred to as an RX buffer. These either exist in device memory where memory read instructions

translate device I/O or the buffer exists in system memory and is asynchronously updated through DMA transactions originating from the device or sensor itself. DMA Transactions are a system level operation with little to no visibility to the Operating system or Hypervisor making this not simply trappable. The memory read accesses are possibly known, atleast to a developer, and possibly to the Operating system, based on how mmap is used if at all within the user space application. If a kernel driver is used, then the ioctl or similar syscall can be addressed as before. Since addresses typically result is a page fault is a trap is desired, the ovrhead can be great. One possible solution is instruction injection by modifying the applications code itself to inject write after read of the data, if those instructions are known to access MMIO regions. This is not an ideal solution since it requires both memory and cache bandwidth and can double the instruction density for functions that require monitoring. A hardware solution such as Intel's Processor Trace could be be used to collect MMIO transactions but the overhead is not bounded and is not fit for real-time applications.

3.3.7 MMIO Replay

Replay of MMIO instructions is a complex topic since they are bounded by physical time based on when the data was last written by an asynchronous entity such as a device or sensor. In replay, physical time is not known and is bounded by logical time such as Lamport clocks or more complex vector clocks. This makes maintaining the memory buffers difficult and instead requires each memory read instruction to be replayed exactly. One solution similar as described for monitoring is instruction injection. In the case of replay, the data is read from a file or buffer in memory, rather than the desired address. This requires basic address checking to make sure the replaying thread has not shifted to a new execution path.

3.4 Measurements

The software for observation used is the PARSEC 3.0 benchmark suite with modifications to identify cycle specific performance and checkpoint latency. These two categories address the overhead of concurrent checkpointing in relation to the response time for each given workload. Additional measurements in the kernel flows are added to better characterize areas of latency such as the soft dirty tracking latency and per thread forced stalls. Non-critical measurements exist in areas that do not directly impact the overhead observed by the workload but can indirectly impact the limits of checkpointing given the total system level performance.

3.4.1 PARSEC Modifications

The PARSEC Benchmarks are organized to separate the initialization from the critical section resulting in a *Region of Interest* or ROI where measurement should take place. Upon entering the ROI, a synchronization is made with CRIU using SIGSTOP and SIGCONT after which synchronization is made with the concurrent checkpoint to begin using a newly added syscall. After these synchronization points, the workload should continue executing be in a controlled environment where all threads are created, file pointers are in place. Memory changes are allowable after initialization through allocation to heap or the increase in stack size from sbrk().

Each workload is broken into cycles that already exist but for sake of measurement are isolated by a new syscall to allow for both event logging and synchronization with the concurrent checkpointing. Synchronization is optional, and only crucial when operating within the bounds of an isochronous scheduling method for responses and checkpoints.

Each benchmark is not created with real-time performance in mind, making each cycle's performance unrelated to all other cycles. Therefor when analyzing performance, the data is represented as a transient. Further analysis of each transient is discussed in the Analysis section.

3.4.2 Important Events and definitions

A list of all events logged by the checkpoint infrastructure is presented in Table 4. These are used to capture the associated activity and support the calculations required to identify various sources of overhead. All events are captured with a time using the RDTSCP instruction.

3.4.2.1 Checkpoint Boundary

The setting of ckpt_inprogress defines a checkpoint boundary from which all future memory writes should be pushed to the following checkpoint period. The state of the application is not synchronized with the ckpt_inprogress flag directly, but rather all future page faults that occur when the flag is set. If pages exist that are writable, the writes to this memory space will be accounted for in the current checkpoint period until the pages are marked read-only and the resulting page fault on a future write will push the writes to the following checkpoint period.

Event Name	Function Location	Important Data
PAGE_FAULT_ANON	do_anonymous_page()	cpu, pid, virtual address
PAGE_FAULT_DIRTY	handle_pte_fault()	cpu, pid, virtual address
PAGE_FAULT_FINISH	handle_pte_fault()	cpu, pid, virtual address
PAGE_FAULT_STALL	handle_mm_fault()	cpu, pid, virtual address
PAGE_FAULT_RESUME	handle_mm_fault()	cpu, pid, virtual address
SWAP_IN	context_switch()	cpu, current pid, parent pid, next pid
SWAP_OUT	context_switch()	cpu, prev pid, parent pid, new pid
SMP_FUNC_CALL_ISSUE	creplay:device_ioctl()	cpu, pid
SMP_FUNC_CALL_COMPLETE	creplay:device_ioctl()	cpu, pid
POP_PAGE	creplay:device_ioctl()	virtual address
PAGE_WALK_START	creplay:device_ioctl()	virtual address
PAGE_WALK_COMPLETE	creplay:device_ioctl()	virtual address
MEMREMAP_START	creplay:device_ioctl()	physical address
MEMREMAP_COMPLETE	creplay:device_ioctl()	physical address, remapped address
PAGE_COPY_START	creplay:device_ioctl()	physical address, remapped address
PAGE_COPY_COMPLETE	creplay:device_ioctl()	physical address, remapped address
CLEAR_DIRTY	creplay:device_ioctl()	virtual address
CHECKPOINT_START	creplay:device_ioctl()	
CHECKPOINT_COMPLETE	creplay:device_ioctl()	
THREAD_HALTED	creplay:device_ioctl()	pid
TRHEAD_UNQUEUED	creplay:device_ioctl()	pid

Table 4. Logging Events

3.4.2.2 Checkpoint Period

The checkpoint period is the window of time between when the cpu register states are collected against a synchronized memory state of the entire process. A checkpoint period represents all state changes that have occurred compared to the previos checkpoint period.

3.4.2.3 Checkpoint Finished

The checkpoint finished event is generated by the kernel driver when a POP_QUEUE command is issued with the ckpt_inprogress flag unset. The event will occur at the end of

the POP_QUEUE flow as to catch copy any pages that cause a Page Fault and waiting for the ckpt_finished_event.

3.4.2.4 Fast Page Fault

This event occurs when a a write operation targets a physical memory page that is clean and marked as read-only. This is the quickest type of page fault which the kernel can quickly update the PTE in the TLB and resume execution. The start of the event is the result of a hardware initiate fault event that is not directly measurable using simple methods. The Fast Page Fault is a product of the dirty page tracking used for concurrent checkpointing and would not occur otherwise.

3.4.2.5 Anonymous Page Fault

Anonymous page faults are a product of writes to virtual memory that is known to a process but does not yet have physical memory associated with it. These faults occur naturally in all operating environments. The dirty page tracking treats these page faults similarly to the fast page fault if the memory type is writable. A read-only page in this case is assumed to be generated by another means that the OS should handle during replay. Like fast page faults, the overhead from anonymous page faults is not easily measured.

3.4.2.6 Page Table Entry Clear (PTE Clear)

Each time a page is saved off for concurrent checkpointing outside of a full process stall, the PTE and TLB entries must be updated to clean and marked as read-only. This requires

locking either the entire page table or a segment of it, possibly stalling other page faults. Since the PTE Clear typically occurs by the checkpointing thread it can add undesired latency to the workload by serialization as a result of these shared locks. The time to perform a PTE Clear by itself does not directly impact the target process.

3.4.2.7 Forced Thread Stall

A forced thread stall occurs when all threads in a process must delay future execution of writes to physical memory. This occurs through either a page fault triggered by the dirty page tracking, syscall synchronization, or smp_func_call methods. Each method of stall is measurable based on the initial event such as the occurrence of the page fault, syscall, or smp_func_call. The stall is observed to finish when a the thread is observed to have been swapped in by a context switch.

3.4.3 Calculations

3.4.3.1 Runtime - $t_{runtime}$

The runtime of the of the workload is the measured time to complete the *Region of Interest* part of the PARSEC benchmark. This is reported out to the console once the benchmark completes.

3.4.3.2 Overhead - $t_{overhead}$

The overhead is the difference in runtime between when concurrent checkpointing is activated using the compilation config *gcc-ckpt_replay* and when it is not using *gcc-hooks*.

3.4.3.3 Page Fault Stall Overhead $t_{pfstall}$

Each page fault stall event can be measured per core by a page fault that occurs when ckpt_inprogress is set. This is measured as the time between PAGE_FAULT_STALL and PAGE_FAULT_RESUME. A summation of all $t_{pfstall}$ will provide a total overhead from page fault stalls represented as $T_{pfstall}$. In the case of multicore systems, this total overhead is shared between all cores. For measurement, the maximum of total observed overhead per each core, is stated as the $T_{pfstall}$.

3.4.3.4 Dirty Tracking Overhead - $T_{tracking}$

With concurrent checkpointing activated the overhead observed from tracking alone is assumed to be $t_{overhead}$ - $T_{pfstall}$. The per page fault overhead $t_{tracking}$ is generalized as $T_{tracking}/n_{pfdirty}$, with $n_{pfdirty}$ as the number of page faults per core.

3.4.4 Non-Critical Measurements

Additional events while measurable, are not as critical to determining the overhead concurrent checkpointing, but rather can be used to determine the limits of the infrastructure and various areas of optimization.

3.4.4.1 Page Copy Performance - t_{page_copy}

The copying a page of memory from a kernel remapped address to a user space buffer using copy_to_user().

3.4.4.2 Page Walk Performance - t_{page_walk}

While the PTE value is commonly known during a page fault, a full page walk is required to collect the various structures in the entire page table to properly lock the page table region from changes and update the dirty tracking. This typically is a long latency serialized operation with little to no cache benefit.

3.4.4.3 Page Remap Performance - t_{page_remap}

The x86 architecture when running in protected mode, the kernel must map physical memory from a user space process into its virtual address space to support access. This is performed by the memremap() using a MEMREMAP_WB flag to generate a write-back configuration.

Chapter 4

RESULTS

4.1 System Setup

The system setup uses a MinnowBoard Turbot Quad-Core board running Ubuntu 18.04 LTS with a modified 4.11.12-rt16+ kernel build booting off a SanDisk 120GB SSD. The kernel config is reduced and scrubbed for compile-in drivers only without any usage of an initramfs. While the system is not heavily tuned for more stringent real-time usage, the "isolcpus=1,2,3" kernel argument is used at boot time to isolate the last two cores from regular system scheduling and interrupts. The PARSEC benchmarks used for measurement and compiled using gcc-hooks and gcc-ckpt_replay. "gcc-ckpt_replay" in this case refers to a modified gcc-hooks configuration with a synchronization for CRIU and syscall that is needed during the initialization of the concurrent checkpoint operation. The PARSEC[1] benchmarks used are blackscholes and canneal. The concurrent checkpointing is run using the configurations for data collection listed in Table 5. These vary the frequency of the checkpoint placement, and PARSEC simulation and native input datasets. The frequency of the POP_QUEUE calls is set to 1/10th of the checkpoint period.

The concurrent checkpoint is first started with CRIU and restored for a clean operating

Frequency	Checkpoint Arguments	Target Datasets
50Hz	./rt_ckpt <..> -s 4096 -f 50	native
25Hz	./rt_ckpt <..> -s 4096 -f 25	native
10Hz	./rt_ckpt <..> -s 4096 -f 10	native
1Hz	./rt_ckpt <..> -s 4096 -f 1	native

Table 5. Caption

environment after which the application will run with concurrent checkpoint active. The output of event logging is redirected to a file.

```
criu dump -t <pid> --shell-job --images-dir criu_images
criu restore --shell-job --images-dir criu_images
./rt_ckpt -m concurrent -o ckpt -p <pid> -s 4096 > concurrent_log.txt
```

Sample output of blackscholes with checkpoint enabled is listed as follows:

```
PARSEC Benchmark Suite Version 3.0-beta-20150206
[HOOKS] PARSEC Hooks Version 1.2
Num of Options: 65536
Num of Runs: 100
Size of data: 2621440
[HOOKS]SIGSTOP Waiting for CRIU
[1]+  Stopped                 taskset -c 2,3 ...
fg
taskset -c 2,3 ./pkgs/apps/blackscholes/inst/amd64-linux.gcc-ckpt_replay/bin/blackscholes 1 ...
[HOOKS]SYSCALL_CKPT Waiting for Concurrent Checkpoint
[HOOKS] Entering ROI
[HOOKS] Leaving ROI
[HOOKS] Total time spent in ROI: 2.647s
[HOOKS] Terminating
```

4.2 Activity Description

The activity of concurrent checkpointing is represented in Figure 4. 3 Areas are selected for viewing. A-B and E-F show the checkpointing process, with the "Checkpoint" signal representing ckpt_inprogress. From these the ckpt_inporogress is set using POP_QUEUE where the dirty tracking queue is emptied and with page walks, page remapping, and page

copying occurring. While the dirty tracking queue is being emptied, a page fault occurs in all three threads causing the page fault stall event. These threads remain in the page fault stall until the ckpt_finished_event is observed by those threads after the ckpt_inprogress signal is unset. For consistency, all pages marked as dirty have a low cost bulk copy that occurs at the end of the checkpoint prior to unsetting ckpt_inprogress. This is to catch any inconsistencies in the timing and sequencing of dirty tracking. In a production enviroment, this can be removed with adequate verification of the dirty tracking sequencing. The performance penalty of the extra copies was tested for and left in for consistency. The actual impact was minimal on the order of less than 10% of $T_{pfstall}$. Small swap events can be seen when the Page Fault stall event occurs but since no other threads are assigned to those cores, the threads remain active, but dormant until the ckpt_finished_event is observed.

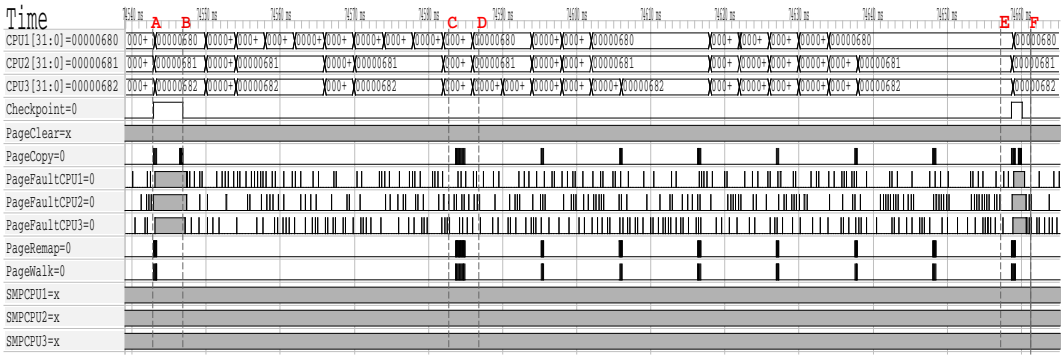
The activity between markers C and D in Figure 4 show the concurrent checkpointing activity. Page faults occur on each core in thier respective threads filling up the dirty queue. A POP_QUEUE command is sent with ckpt_inprogress unset to collect the dirty page thus far but not yet prepare for a checkpoint. The page walks, remaps, and copies during this phase have little to no impact on the target process. Serialization of page faults, and some delays can be observed due to the use of locks in the page table structure, and the dirty queue. Global locks are avoided to prevent latency propagation from the checkpointing facility to the target thread.

The required use of smp_func_call is demonstrated in Figure 5 when the GET_REGS command occurs prior to the threads being halted through either a page fault stall or scheduler swap. Since a full context_switch is not observed, the swap events do not show in the wave form, but this is the point the cpu state registers are captured. A page fault stall event is still observed to prevent writes from entering the previous checkpoint period but this has no impact on the cpu state registers used for the checkpoint. The final page walk and remap

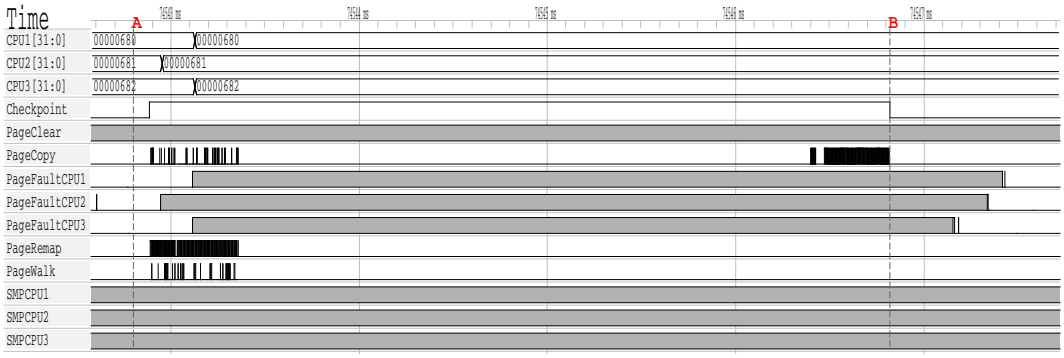
is to have a second check of the address faults that occur after the GET_REGS command, but the contents to be written to memory are not allowed until the page fault resumes after ckpt_inprogress is unset.

4.3 Blackscholes Data

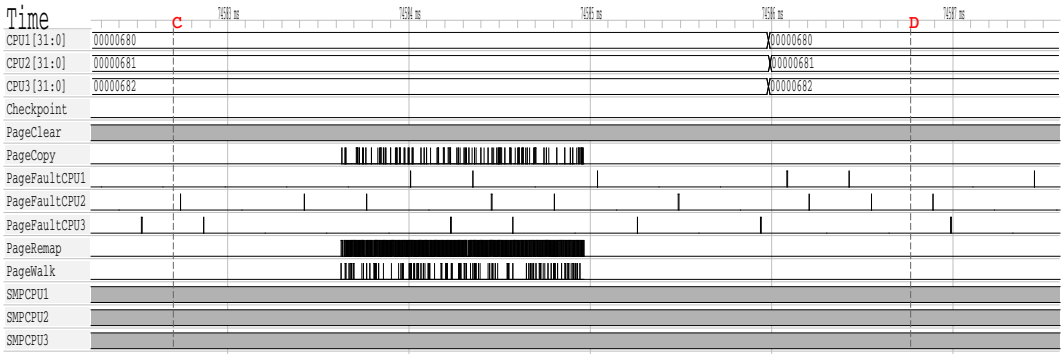
The runtime data comparing checkpoint frequency to standalone application without checkpointing is presenting in Table 6. The overhead observed by looking at the run-time is quite small with the most observed as 1.059 slowdown with a checkpoint target frequency of 50Hz in the 2 thread configuration. The checkpoint frequency of 50Hz is not sustainable by looking at the activity waveform in Figure 6 and comparing it against the equivalent 25Hz configuration. This shows that the Succeeding POP_QUEUE command is delayed by about 60% of the checkpoint period since the user space checkpoint application is busy saving the state changes form the prior checkpoint. Additional measurements can be made with the user space checkpoint application to find it's limits and possible optimization's to support higer frequencies, but this work is focused on the target application overhead from concurrent checkpointing. A breakdown of the measurable overhead and assumed dirty tracking overhead is presented in Table 7. The dirty tracking overhead is not easily measured since it requires visibility of hardware events such as interrupts and micro-architectural latency. For analysis, this overhead will be represented as the remaining overhead after subtracting the total delay observed from page fault stalls. Further study of the checkpoint overhead can be used to define a overhead per page fault for the tracking and a forced stall overhead. The per page fault overhead uses the assumed dirty tracking overhead divided by the number of page faults resulting in a rough overhead in microseconds. For Blackscholes, these items are calculated from the data in Table 8.



(a) Wide View



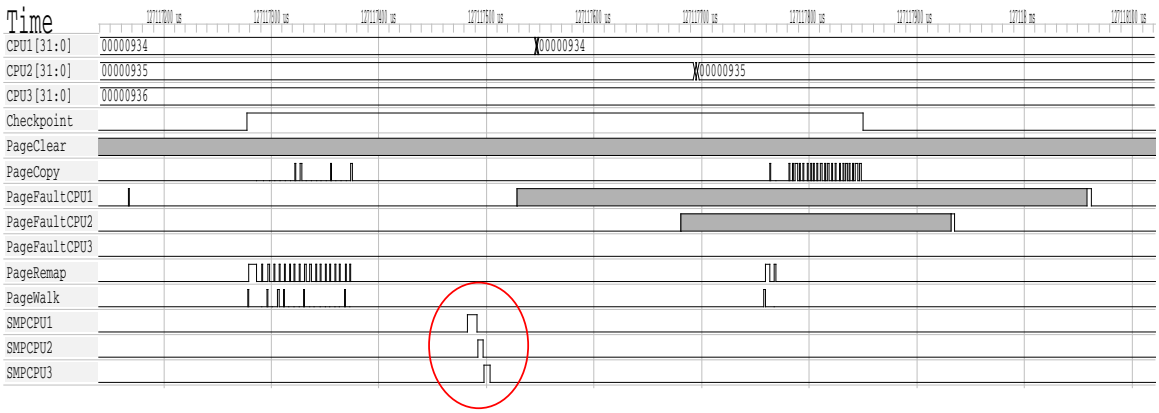
(b) A-B



(c) C-D

(d) E-F

Figure 4. 3 Thread Concurrent Checkpoint Activity



(a) SMP_FUNC_CALL Usage

Figure 5. 3 Thread Concurrent Checkpoint Activity - Cont.

	Execution times (seconds)					Slowdown factors			
Dataset	no chkp	1Hz	10Hz	25Hz	50Hz	1Hz	10Hz	25Hz	50Hz
1 Thread	329.7	335.125	339.826	341.557	340.401	1.016	1.031	1.037	1.032
2 Thread	164.592	167.355	170.357	170.388	174.365	1.017	1.035	1.035	1.059
3 Thread	109.928	111.811	114.388	115.923	113.038	1.017	1.041	1.055	1.028

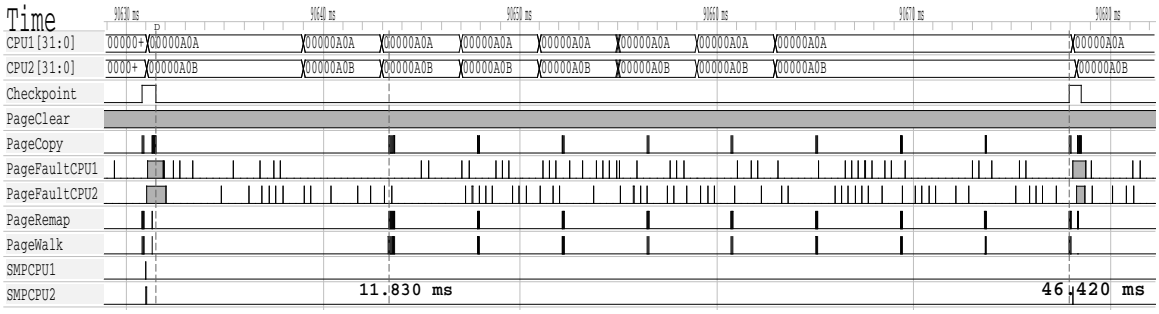
Table 6. Execution times of Blackscholes benchmark program

Execution overhead in ms				
Results	Total overhead	Page Fault Stalls (%)	Assumed Dirty Tracking (%)	Per Checkpoint Overhead(ms)
1 Thread 1Hz	5.425	2.709(49.9%)	2.715(50.0%)	8.797
1 Thread 10Hz	10.126	5.916(58.4%)	4.208(41.6%)	1.938
1 Thread 25Hz	12.252	8.481(69.2%)	3.751(30.6%)	1.149
1 Thread 50Hz	10.701	6.386(59.7%)	4.254(39.8%)	0.445
2 Thread 1Hz	2.763	1.490(53.9%)	1.273(46.1%)	9.430
2 Thread 10Hz	5.765	3.598(62.4%)	2.165(37.6%)	2.239
2 Thread 25Hz	5.796	2.625(45.3%)	3.155(54.4%)	0.658
2 Thread 50Hz	9.773	5.872(60.1%)	3.862(39.5%)	0.770
3 Thread 1Hz	1.883	0.739(39.2%)	1.144(60.7%)	6.657
3 Thread 10Hz	4.460	2.867(64.3%)	1.592(35.7%)	2.533
3 Thread 25Hz	5.995	4.291(71.6%)	1.693(28.2%)	1.556
3 Thread 50Hz	3.110	1.340(43.1%)	1.741(56.0%)	0.255

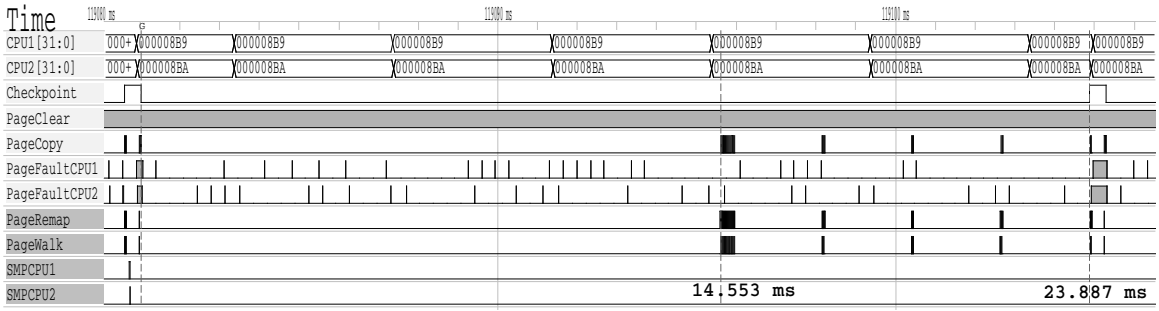
Table 7. Blackscholes Overhead

Execution overhead in ms					
Results	$n_{pfdirty}$	$n_{checkpoint}$ (Actual Frequency)	$\frac{n_{pfdirty}}{sec}$	$\frac{n_{pfdirty}}{n_{checkpoint}}$	$t_{pfdirty}$(us)
1 Thread 1Hz	843433	308(0.9)	2516.771	2738.419	3.219
1 Thread 10Hz	981168	3052(9.0)	2887.266	321.484	4.289
1 Thread 25Hz	1016093	7383(21.6)	2971.449	137.626	3.691
1 Thread 50Hz	984006	14343(42.1)	2890.726	68.605	4.323
2 Thread 1Hz	410896	158(0.9)	2455.236	2600.608	3.688
2 Thread 10Hz	491228	1607(9.4)	2883.521	305.680	4.592
2 Thread 25Hz	498993	3987(23.4)	2928.569	125.155	6.761
2 Thread 50Hz	403250	7622(43.7)	2312.677	52.906	10.213
3 Thread 1Hz	204176	111(1.0)	1826.082	1839.423	6.176
3 Thread 10Hz	325558	1132(9.9)	2846.085	287.595	5.558
3 Thread 25Hz	243136	2757(23.8)	2097.392	88.189	7.198
3 Thread 50Hz	197945	5263(46.6)	1751.137	37.611	9.314

Table 8. Blackscholes Overhead Analysis



(a) 2 Thread 25Hz



(b) 2 Thread 50Hz

Figure 6. Blackscholes: Comparing 2 Thread Checkpoint Period 25Hz to 50Hz

	Execution times (seconds)					Slowdown factors			
Dataset	no chkp	1Hz	10Hz	25Hz	50Hz	1Hz	10Hz	25Hz	50Hz
1 Thread	354.742	361.801	386.79	397.094	390.152	1.020	1.090	1.119	1.100
2 Thread	184.859	190.196	193.339	197.084	204.042	1.029	1.046	1.066	1.104
3 Thread	127.815	132.57	135.818	139.036	151.94	1.037	1.063	1.088	1.189

Table 9. Execution times of Canneal benchmark program

4.4 Canneal Data

Canneal has larger memory footprint than blackscholes with irregular memory access. The checkpoint overhead is more apparent than blackscholes, specifically seeing that the number of page faults is 4x greater for canneal in 1 thread configurations at 10hz than the similar configuration for blackscholes. The slow down is greater for this reason as well the overhead observed from dirty tracking, vs overhead from page fault stalls. 9. Serialization is also more apparent from page faults in the 2 and 3 thread configurations. Canneal requires more time to save larger amounts of state changes due to the greater number of page faults. Since the $t_{pfdirty}$ does not match blackscholes closely, the concurrent checkpointing was tested with only POP_QUEUE commands being issues and never setting ckpt_inprogress. This data is representing in Tables 12 and 13. The overhead is isolated per core to see possible variation between threads on each core. Looking at the waveform activity in Figure 7, the extra pagefaults that occur on Core 1 are skewing the overhead results, making Core 2 and 3 numbers appearing to have a larger overhead. for this reason all data captured focuses on Core 1 results only ignoring data for Core 2 and 3 in the case of multicore runs.

Execution overhead in ms				
Results	Total overhead	Page Fault Stalls (%)	Assumed Dirty Tracking (%)	Per Checkpoint Overhead(ms)
1 Thread 1Hz	7.059	1.760(24.9%)	5.299(75.1%)	5.569
1 Thread 10Hz	32.048	13.804(43.1%)	18.244(56.9%)	4.086
1 Thread 25Hz	42.352	21.543(50.9%)	20.806(49.1%)	2.549
1 Thread 50Hz	35.410	20.077(56.7%)	15.248(43.1%)	1.255
2 Thread 1Hz	5.337	2.075(38.9%)	3.262(61.1%)	11.465
2 Thread 10Hz	8.480	2.985(35.2%)	5.487(64.7%)	1.712
2 Thread 25Hz	12.225	3.916(32.0%))	8.282(67.7%)	0.866
2 Thread 50Hz	19.183	8.269(43.1%)	10.854(56.6%)	0.899
3 Thread 1Hz	4.755	1.144(24.1%)	3.611(75.9%)	9.081
3 Thread 10Hz	8.003	2.464(30.8%)	5.535(69.2%)	1.954
3 Thread 25Hz	11.221	4.184(37.3%)	7.019(62.5%)	1.289
3 Thread 50Hz	23.379	14.197(60.7%)	9.141(39.1%)	2.248

Table 10. Canneal Overhead

Execution overhead in ms					
Results	n_{pfdirty}	$n_{\text{checkpoint}}$ (Actual Frequency)	$\frac{n_{\text{pfdirty}}}{sec}$	$\frac{n_{\text{pfdirty}}}{n_{\text{checkpoint}}}$	t_{pfdirty}(us)
1 Thread 1Hz	1059333	316(0.9)	2927.944	3352.320	5.002
1 Thread 10Hz	4067596	3378(8.7)	10516.290	1204.143	4.485
1 Thread 25Hz	4094040	8452(21.3)	10310.002	484.387	5.082
1 Thread 50Hz	1922762	15993(41.0)	4928.238	120.225	7.930
2 Thread 1Hz	564792	181(1.0)	2969.526	3120.398	5.808
2 Thread 10Hz	722335	1743(9.0)	3736.106	414.421	7.826
2 Thread 25Hz	736951	4523(22.9)	3739.274	162.934	11.471
2 Thread 50Hz	723334	9199(45.1)	3545.025	78.632	15.648
3 Thread 1Hz	408758	126(1.0)	3083.337	3244.111	8.984
3 Thread 10Hz	511564	1261(9.3)	3766.541	405.681	11.060
3 Thread 25Hz	544793	3247(23.4)	3918.359	167.783	13.258
3 Thread 50Hz	536797	6315(41.8)	3550.386	85.003	17.841

Table 11. Canneal Overhead Analysis

	Execution times (seconds)				
Dataset	no chkp	1Hz	10Hz	25Hz	50Hz
3 Thread	127.815	130.383	131.572	133.204	133.201

Table 12. Execution times of Canneal with dirty tracking only

Execution overhead in ms			
Results	$n_{pfdirty}$	$\frac{n_{pfdirty}}{sec}$	$t_{pfdirty}(us)$
3 Thread 1Hz Core 1	391461	3002.393	6.560
3 Thread 10Hz Core 1	563240	4280.850	6.670
3 Thread 25Hz Core 1	654184	4911.144	8.238
3 Thread 50Hz Core 1	573570	4306.049	9.390
3 Thread 1Hz Core 2	210927	1617.749	12.175
3 Thread 10Hz Core 2	350625	2664.891	10.715
3 Thread 25Hz Core 2	432056	3243.566	12.473
3 Thread 50Hz Core 2	343797	2581.039	15.666
3 Thread 1Hz Core 3	208725	1600.861	12.303
3 Thread 10Hz Core 3	351888	2674.490	10.677
3 Thread 25Hz Core 3	431464	3239.122	12.490
3 Thread 50Hz Core 3	342478	2571.137	15.727

Table 13. Canneal Overhead Analysis

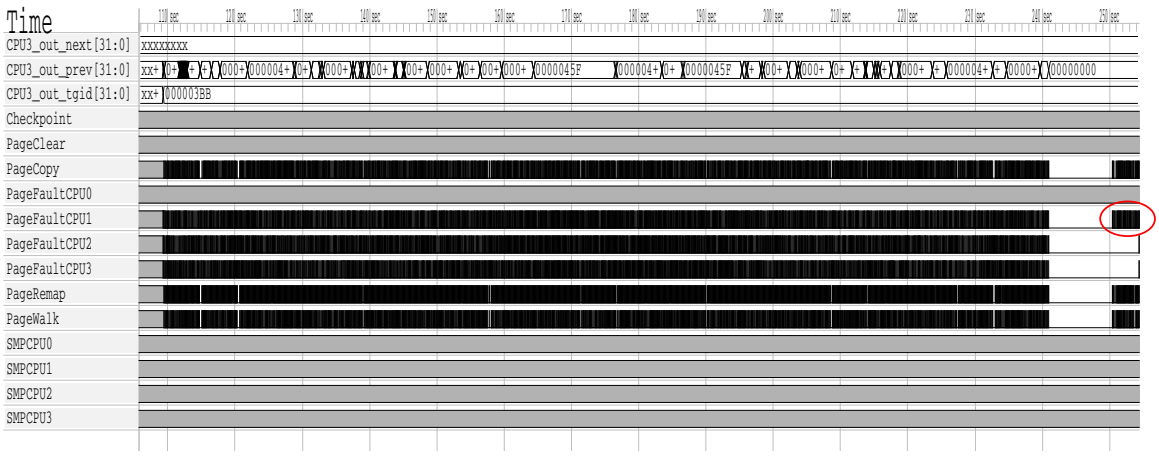


Figure 7. Canneal: 3 Thread 10 Hz Dirty Tracking Only
Since all the threads join prior to the end of checkpointing, the data for Core 1 has more pagefaults causing the overhead to be skewed

4.5 Analysis

For the two workloads with varying configurations there is a trend between checkpoint frequency and overhead observed. The goal of this research is to de-emphisize the dominant delay from the checkpoint itself and shift it into a concurrent operation where only limited amount of the overhead is observed by the target application. This has been successful in showing that less that 75% of the total overhead is a directly related to stopping the process while the application state is captured. In most cases, there is a even distribution of forced stalls, and general overhead with a few observances of tracking being a major overhead. In blackscholes, most checkpoint delays are observed to be less than 5ms when running in multicore operation and in single core mode, most checkpoints were 500us or less. This validates the solution as a candidate for soft real-time systems that can absorb a average 10% overhead to the required response time. When first developing the checkpoint, all page copies within the checkpoint flow would perform a full page walk, remap, and finalize with unmap forcing the operation to repeat completely. This greatly increased the checkpoint time since each page walk took on average 1us and the remapping from physical to virtual address in the kernel took on average 3.5us. The page copying itself can be lengthy when cache misses occur, but averages out to less than 1us. The page walk and remapping steps can be eliminated if caching the result and leaving the remapping in place for future events. This results in a rather difficult cleanup from an OS perspective once checkpointing is finished and the application either continues normally, or exits. The data from the tables are compiled into graphs in Figure 8 and Figure 9 to assist in visualizing trends between checkpoint frequency and overhead measurements. Figure 9(a) highlights an average overhead for each page fault due to dirty tracking. Figure 9(b) shows the comparison of tracking overhead with and without a checkpoint occurring. Figure 9(c) represents the number of page faults due to tracking. The

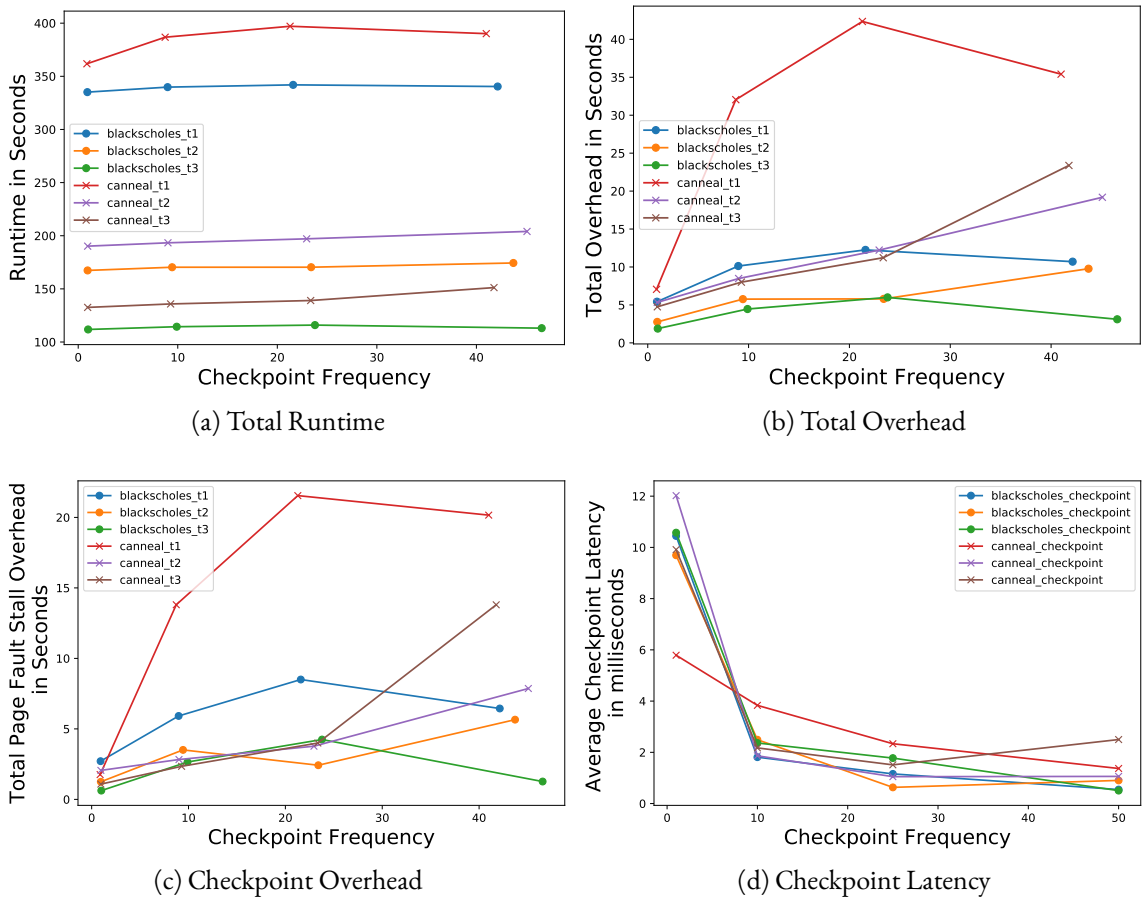
checkpoint frequency in this setup has less impact on number of page faults it produces since as the frequency increases less POP_QUEUE commands are possible within the checkpoint period. Figures 10 show a histogram of the amount of repeated unique addresses during a checkpoint period due to the POP_QUEUE command. For both workloads running in a 3 thread configuration, the amount of repeated addresses increase exponentially as the checkpoint frequency reduces. This is why in Figure 8(b) an unusually large overhead is observed for canneal running 1 thread. When a checkpoint period is larger such as 15 or more, it is beneficial to cluster the POP_QUEUE commands near the end of the checkpoint period, rather than spread them out evenly as done by these tests. The workloads used parallelize well across all cores, allowing the page faults to also be spread evenly across all cores. For workloads that are parallelized based on producer and consumer, the producer threads will observe the bulk of the tracking overhead and consumer thread will be generally left unaffected. As a workload becomes more parallel across multiple cores, it is possible for page faults to serialize due to the semaphore for the mm_struct, mmap_sem. This is required to be held for most page faults, and similarly is required in the POP_QUEUE command flow. This fits well with page fault overhead doubling when parallelism is used. Reducing the dependency on the mmap_sem may reduce this latency, but it may not be possible with the current page fault architecture. For comparison with CRIU, six checkpoints are measured for both Canneal and Blackscholes when running in the 3 Thread gcc-hooks configuration presented in Table 14. Checkpoints performed by CRIU, copy the entire memory space which is why the checkpoint time is very large. Criu's page copy routines are quite inefficient in comparison. In [4], using CRIU's pre-dump can further improve the checkpoint times into the 100s of milliseconds range, but the work presented here presents a checkpoint method 2 orders of magnitude faster than [4].

Total Checkpoint Time(s)	Page Copy Time(s)	Core State Copy Time(ms)	Memory Copy Size(MB)	Average 4K Page Copy Time(us)
Blackscholes 3 Thread				
5.77s	5.71s	0.36ms	610.00MB	36.6us
6.28s	6.24s	0.34ms	610.00MB	39.9us
5.43s	5.35s	0.46ms	610.00MB	34.3us
5.46s	5.43s	0.47ms	610.00MB	34.8us
6.42s	6.39s	0.37ms	610.00MB	40.9us
5.73s	5.69s	0.52ms	610.00MB	36.4us
Canneal 3 Thread				
14.32s	14.28s	1.43ms	847.00MB	65.8us
19.29s	19.25s	0.47ms	847.00MB	88.8us
21.84s	21.82s	0.44ms	847.00MB	100.6us
19.04s	18.99s	0.38ms	847.00MB	87.5us
25.51s	25.46s	0.45ms	847.00MB	117.4us
31.49s	31.46s	0.47ms	847.00MB	145.0us

Table 14. CRIU Checkpoint Comparison Data

4.5.1 Concurrent Checkpoint Applied Analysis

The purpose of concurrent checkpoint with respect to Real-Time Embedded Systems is to create a reliable checkpoint frame work with minimal impact to the performance of a target Real-Time application. Such systems have a desired response time within a defined Quality of Service. The checkpoint facility can be orchestrated differently based on the allowable overhead and any possible discoveries during its operation. The simplest solution is to alter the frequency and placement of the POP_QUEUE requests to reduce the dirty tracking overhead if the same addresses are known to be found multiple times. This can be reduced by delaying the POP_QUEUE longer after a ckpt_inprogress is unset from the previous checkpoint period. As noticed when attempting to checkpoint the benchmarks at 50Hz, the POP_QUEUE commands are not equally placed and are usually delayed by the amount of time to save the previous checkpoint to disk. This shows that while the check-



(a) Total Runtime

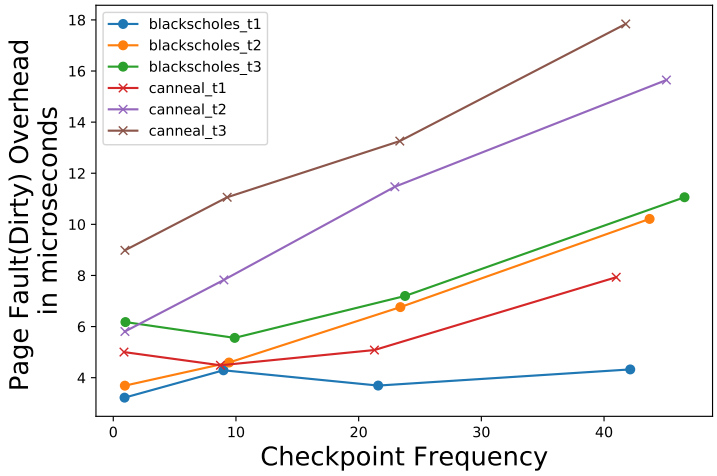
(b) Total Overhead

(c) Checkpoint Overhead

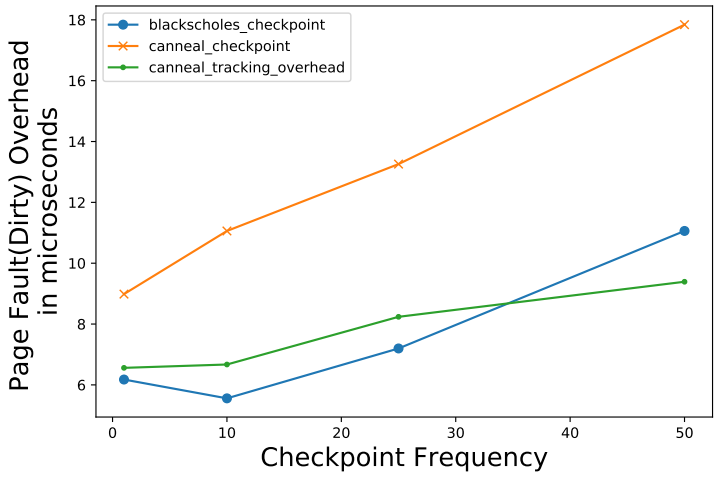
(d) Checkpoint Latency

Figure 8. Checkpoint Latency Analysis Graphs

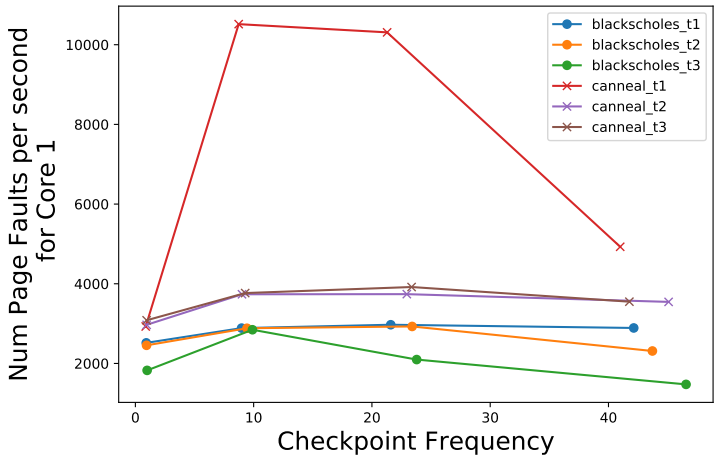
pointing frequency increases, the amount of page faults for dirty tracking do not increase. The tracking overhead similarly does not increase at the same rate as the checkpointing frequency either. Additional hooks can be added to reduce dirty tracking overhead for common duplicates such as stack memory. A *stage* is attached to each vma_struct to provide this per page classification. A requeue counter is present used to count the amount of repeats of a specific page. Neither the stage or counter is not used in the current implementation. This provides extensible in both the kernel modifications, or the user space interface to manipulate the behavior as needed.



(a) Tracking Overhead

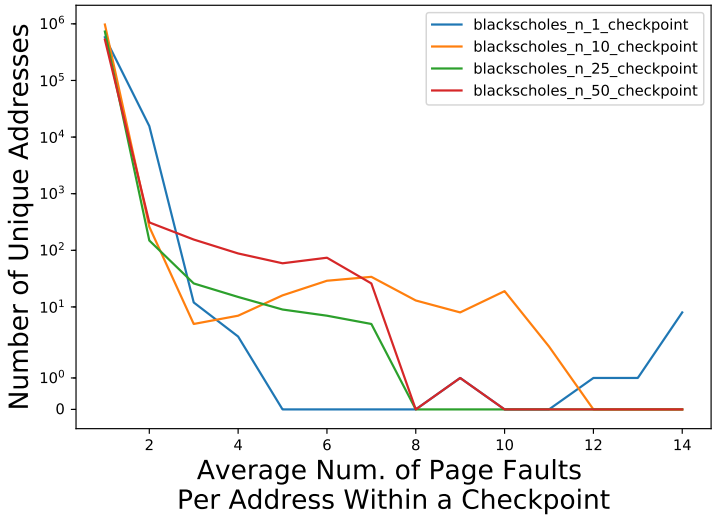


(b) Comparing Tracking Overhead without Checkpointing (3 Thread)

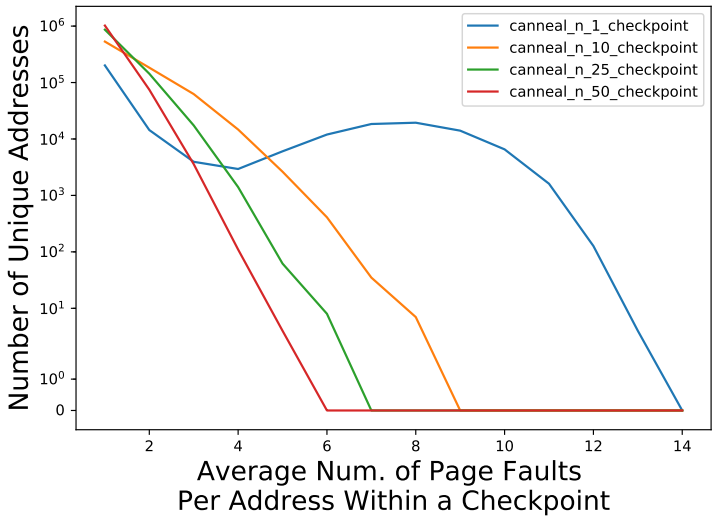


(c) Page Fault Occurrences

Figure 9. Tracking Overhead Analysis Graphs



(a) Blackscholes 3 Threads



(b) Canneal 3 Threads

Figure 10. Histograms of Repeated Page Faults

Chapter 5

CONCLUSION

The method presented in this work is a low cost solution to perform concurrent checkpointing by using a dirty page tracking scheme for soft real-time embedded environments. The method performs best when memory is heavily reused during the applications life to further reduce the per page fault cost. If incorporated into CRIU to fully utilize all its capabilities along side concurrent checkpointing, a robust checkpoint and replay can be produced and dramatically reduce the checkpoint delay spent in copying application memory state. Concurrent checkpointing can quickly exhaust a storage medium as well depending on the frequency of checkpoints and amount of dirty pages per checkpoint. By employing various algorithms to deciding which pages to copy concurrently or during a checkpoint can help in reducing the dirty tracking overhead. The highest cost of a checkpoint comes from remapping the physical memory from one process into the kernel before copying into a storage medium. The actual copy of the memory is quite low cost compared to other latency's by an order of magnitude. One such comparison is to not copy memory and leave it marked as dirty, but perform all the remapping during each checkpoint period. This makes the checkpoint more difficult to schedule since a page fault stall is what helps halt all threads during a checkpoint and it is difficult to achieve checkpoint synchronization if writes are not restricted around the checkpoint boundary. Additional optimization could be made to reduce dirty tracking costs for stack memory, but would require understanding where the stack pointer is, especially in cases where an applications stack grows very large with more data items in stack than heap.

The ideal usage for this method of concurrent checkpointing exists at 10Hz or below

due to some of the long latency's observed in the microseconds. When stepping into 100Hz or 1KHz of checkpointing, the storage medium will quickly fill, and the overhead of checkpointing will become a significant portion of the overall compute on the system. For such requirements, a hardware assisted checkpointing solution would be appropriate. With an observed maximum of 1.189x(Canneal 3 Thread 50hz) slowdown for checkpointing, and a more common 1.05x slowdown this solution can be used without impacting the application's quality of service. The additional system resources and compute power required greatly depends on the application in question. When comparing the current cost per core and cost for additional memory versus that of a semi custom design with an external hardware based monitoring device, this software based method will be a more likely solution.

REFERENCES

[1] Christian Bienia. "Benchmarking Modern Multiprocessors". PhD thesis. Princeton University, Jan. 2011.

[2] Pat Brouillette and Jason Roberts. "Real-Time Debugging with SVEN and OMAR". In: *Intel® Technology Journal* 16.1 (2012), pp. 62–73. URL: https://software.intel.com/sites/default/files/article/380173/real-time-debugging-with-sven-inteltechjournal.pdf.

[3] N. Chen and S. Ren. "Building a Coordination Framework to Support Behavior-Based Adaptive Checkpointing for Open Distributed Embedded Systems". In: *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*. Jan. 2007, 257b–257b. DOI: 10.1109/HICSS.2007.114.

[4] X. Chen, J. H. Jiang, and Q. Jiang. "A Method of Self-Adaptive Pre-Copy Container Checkpoint". In: *2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC)*. Nov. 2015, pp. 290–300. DOI: 10.1109/PRDC.2015.11.

[5] *CoreSight Architecture Specification*. ID092613. Version 2.0. ARM. 2013.

[6] CRIU. *Checkpoint/restore in userspace*. Accessed: 2017. URL: https://criu.org/.

[7] Joseph Devietti et al. "DMP: Deterministic Shared Memory Multiprocessing". In: *SIGPLAN Not.* 44.3 (Mar. 2009), pp. 85–96. DOI: 10.1145/1508284.1508255. URL: http://doi.acm.org/10.1145/1508284.1508255.

[8] J Duell, Paul Hargrove, and Eric Roman. "The Design and Implementation of Berkeley Lab"s Linux Checkpoint/Restart". In: *LBNL Technical Report, LBNL 54941* (Jan. 2003).

[9] Linux Foundation. *Project ACRN*. Accessed: 2018. URL: https://projectacrn.org.

[10] *Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 3B*. 253669-061US. Intel. 2016.

[11] K. Li, J. F. Naughton, and J. S. Plank. "Real-time, Concurrent Checkpoint for Parallel Programs". In: *SIGPLAN Not.* 25.3 (Feb. 1990), pp. 79–88. DOI: 10.1145/99164.99173. URL: http://doi.acm.org/10.1145/99164.99173.

[12] A. Miraglia et al. "Peeking into the Past: Efficient Checkpoint-Assisted Time-Traveling Debugging". In: *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. Oct. 2016, pp. 455–466. DOI: 10.1109/ISSRE.2016.9.

[13] Pablo Montesinos, Luis Ceze, and Josep Torrellas. "DeLorean: Recording and Deterministically Replaying Shared-Memory Multiprocessor Execution Ef?Ciently". In: *SIGARCH Comput. Archit. News* 36.3 (June 2008), pp. 289–300. DOI: 10.1145/1394608.1382146. URL: http://doi.acm.org/10.1145/1394608.1382146.

[14] S. Narayanasamy, G. Pokam, and B. Calder. "BugNet: continuously recording program execution for deterministic replay debugging". In: *32nd International Symposium on Computer Architecture (ISCA'05)*. June 2005, pp. 284–295. DOI: 10.1109/ISCA.2005.16.

[15] S. Neogy, A. Sinha, and P. K. Das. "Checkpoint processing in distributed systems software using synchronized clocks". In: *Proceedings International Conference on Information Technology: Coding and Computing*. Apr. 2001, pp. 555–559. DOI: 10.1109/ITCC.2001.918855.

[16] Robert O'Callahan et al. "Engineering Record and Replay for Deployability". In: *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*. USENIX ATC '17. Santa Clara, CA, USA: USENIX Association, 2017, pp. 377–389. URL: http://dl.acm.org/citation.cfm?id=3154690.3154727.

[17] *rr: lightweight recording and deterministic debugging*. Accessed: 2017. URL: http://rr-project.org/.

[18] Siemens. *Jailhouse: Linux-based partitioning hypervisor*. Accessed: 2018. URL: https://github.com/siemens/jailhouse.

[19] *Software Development Kit for the System Visible Event Nexus Technology (SVEN)*. 328506-001US. Rev 1.0. Intel. 2013.

[20] J. J. P. Tsai, K. Y. Fang, and Y. D. Bi. "On real-time software testing and debugging". In: *Proceedings., Fourteenth Annual International Computer Software and Applications Conference*. 1990, pp. 512–518. DOI: 10.1109/CMPSAC.1990.139423.

[21] J. J. P. Tsai et al. "A noninterference monitoring and replay mechanism for real-time software testing and debugging". In: *IEEE Transactions on Software Engineering* 16.8 (Aug. 1990), pp. 897–916. DOI: 10.1109/32.57626.

[22] D. Vogt et al. "Lightweight Memory Checkpointing". In: *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. June 2015, pp. 474–484. DOI: 10.1109/DSN.2015.45.

[23] Ying Zhang and Krishnendu Chakrabarty. "Energy-aware adaptive checkpointing in embedded real-time systems". In: *Proceedings of the conference on Design, Automation and Test in Europe-Volume 1*. IEEE Computer Society. 2003, p. 10918.