2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)

Prosper: Program Stack Persistence in Hybrid Memory Systems

Arun KP
Indian Institute of Technology
Kanpur, India
Email: kparun@cse.iitk.ac.in

Debadatta Mishra
Indian Institute of Technology
Kanpur, India
Email: deba@cse.iitk.ac.in

Biswabandan Panda
Indian Institute of Technology
Bombay, India
Email: biswa@cse.iitb.ac.in

***Abstract*—A persistent and crash-consistent execution state is essential for systems to guarantee resilience against power failures and abrupt system crashes. The availability of non-volatile memory (NVM) with read/write latency comparable to DRAM allows designing efficient checkpoint mechanisms for process persistence. Operating system (OS) level checkpoint solutions require capturing the change in the execution state of a process in an efficient manner. One of the crucial components of the execution state of any process is its memory state consisting of mutable stack and heap segments. Tracking modifications to the program stack is interesting because of its unique grow/shrink usage pattern and activation record write characteristics. Moreover, the stack is used in a programmer-agnostic manner where the compiler makes use of the support provided by the underlying ISA to use the stack and the OS manages the memory used by the stack region in an on-demand fashion.**

In this paper, we show the benefit of a checkpoint-based mechanism for stack persistence and the inefficiency of adapting existing generic memory persistence mechanisms for the stack region. We propose *Prosper*, a hardware-software (OS) co-designed checkpoint approach for stack persistence. *Prosper* tracks stack changes at sub-page byte granularity in hardware, allowing symbiosis with OS to realize efficient checkpoints of the stack region. *Prosper* significantly reduces (on average ∼4×) the amount of data copied during checkpoint and improves the overall checkpoint time with minimum overhead (less than 1% on average). Integration of *Prosper* with existing state-of-the-art memory persistence mechanisms (such as SSP) for heap provides 2.6× improvement over solely using the state-of-the-art mechanism for the entire memory area persistence.

I. Introduction

Process persistence using checkpoint techniques [17], [25], [31] has gained popularity with the emergence of hybrid memory systems consisting of traditional volatile random access memory (DRAM) and byte-addressable non-volatile memory (NVM).

To achieve process persistence through checkpoints, it is required to persist the process state periodically. The process state consisting of the CPU register state, memory state, and other associated states should be checkpointed in a manner such that the process can resume from the last execution point across system restarts [48]. Capturing periodic snapshots of the memory state of processes consisting of different mutable memory segments (e.g., heap and stack) present non-trivial challenges, both in terms of checkpoint complexity and checkpoint size [47]. Therefore, many research contributions treat the general problem of persisting the memory state in a consistent manner in isolation by employing techniques at both the software layers [11], [15], [19], [24], [30], [32] and at the hardware layer [1], [8], [41], [56].

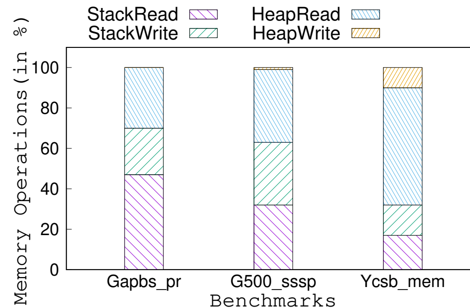


Fig. 1: Memory operations to the stack and heap regions (in %) demonstrating the significance of stack operations.

In the context of process persistence, the OS-level checkpoint solutions are more practical compared to the generic memory persistence solutions considering the semantic proximity of the OS to the notion of processes. For example, it is non-trivial for a hardware or user space memory persistence technique to demarcate the boundaries for the memory state of a process spanning across the user and OS layers. On the other hand, OS-level checkpoint procedures can potentially leverage the additional hardware support for memory persistence (with some adaptation) to simplify the complexities associated with periodic memory checkpoints. In this paper, we demonstrate that the state-of-the-art solutions *are not suitable for all memory segments*, specifically for memory segments with special characteristics such as *the program stack*. Furthermore, we propose *specialized hardware extensions* to efficiently checkpoint the stack region and show that it can improve the overall efficiency of OS-level checkpoint solutions when combined with tailored generic memory persistence techniques.

Typically, the size of the stack segment is smaller than the heap segment, but *the number of operations* on the stack can be significant for some applications. Figure 1 shows the fraction of memory operations in the stack region for three representative benchmarks from the graph and cloud workloads—Gapbs_pr [5], G500_sssp [39], and YCSB [13]. We traced these benchmarks for stack and heap operations using Intel Pin [37] on a four-core Intel(R) Xeon(R) W-

TABLE I: Comparison of existing memory persistence mechanisms.

Property	SSP [41]	JUSTDO [24]	SoftWrAP [19]	Timestone [30]	Romulus [15]
Achieves process persistence	✗	✗	✗	✗	✗
Works without compiler support	✓	✗	✗	✗	✗
Stack pointer awareness	✗	✗	✗	✗	✗
Allows stack in DRAM	✗	✗	✓	✗	✗

2104 system for the highest weighted interval identified by SimPoint [23]. For Gapbs_pr, 70% operations (reads and writes) are performed to the stack regions. We highlight some of the important usage characteristics of the stack segment (used to implement function calls and store program objects in local scope) that differ from other memory segments before discussing the applicability of state-of-the-art techniques.

Usage pattern. The stack exhibits a grow and shrink pattern, i.e., back-and-forth movement of the stack pointer (SP) during the lifetime of processes (and threads), which is different from the allocate-use-free pattern of other segments such as heap.

Write characteristics. The stack region is not only write-intensive (Figure 1), but also maintains activation records across function invocations and returns, resulting in a significant number of writes to a cluster of memory addresses.

Indirect usage. Unlike heap, where the application layer uses the region through explicit allocation and de-allocation, for a stack, the compiler or run-time system introduces the required stack operations that are hidden from the application layer. The role of the OS is a little different for the stack as it handles the growth and shrinkage of a stack in an on-demand fashion [3].

For stack persistence, periodic commit-based techniques *operating in tandem* with application execution [8], [19], [56] may give rise to inefficiencies because they can not adequately address the subtleties of stack usage. *First*, the stack usage pattern may lead to unnecessary operations because the SP may grow and shrink during an interval. We show that having future knowledge regarding the value of SP at the points of commits (referred to as *SP awareness*) significantly improves the efficiency of existing techniques (Section II-A). We designate a memory persistence mechanism to be *SP aware* if the overhead incurred to persist the stack region is predominantly determined by the active stack region at the commit point. *Second*, considering the write-intensive nature of the stack region, maintaining the stack in NVM leads to performance and endurance issues [54], [55]. Approaches that do not employ periodic checkpoints have to maintain the stack in NVM along with the meta-data required to achieve consistency. *Third*, many existing approaches, specifically the logging-based approaches—redo, undo [4], [52] and their variations [24])—require invocation of special APIs from the application layer for different events such as load, store, and commit. Considering the indirect usage of the stack region, it requires non-trivial extensions to the compiler to insert calls appropriately for different operations in the stack region.

Checkpoint-based solutions allow allocation and usage of the stack in DRAM while achieving persistence by copying the dirtied stack memory addresses into NVM at the end of every checkpoint interval. An OS-level periodic checkpoint solution for the stack region can address the previously mentioned challenges for the following reasons. *First*, the checkpoint mechanism is *SP aware* as the activity performed by OS at the time of checkpoint (i.e., copying the dirtied stack memory into NVM) depends upon the *active* stack region(s). *Second*, hosting the stack region in DRAM alleviates the problem of excessive writes to NVM. Moreover, periodic checkpoints allow higher levels of write coalescing, addressing the inefficiency concerns due to the *write characteristics*. *Third*, an OS-layer checkpoint solution for stack regions can be used in a generic manner without requiring any special support from the compiler/run-time addressing the challenges arising due to the *indirect usage* of the stack memory. One of the challenges in capturing the snapshot for the stack region is amplification of checkpoint size due to limited hardware support for efficient dirty tracking. For example, as we show in Section II-B, dirty tracking of the stack region at the OS page granularity (e.g., SoftDirty [18], LDT [45]) results in significantly large checkpoint sizes compared to dirty tracking at the sub-page granularity.

Observations. Without OS-level adaptations, existing memory persistence techniques *in their current form* are not adequate to achieve efficient process persistence. Even with OS-layer adaptations, there can be inefficiencies when existing techniques are used for stack considering the usage and access pattern of the stack region. A summary of existing techniques along with their applicability is presented in Table I. While the OS-layer checkpoint approach for the stack can be seen as an extension to checkpointing other non-memory states of the process (e.g., the register state), the checkpoint overhead due to the stack modifications should be minimized. Moreover, generic hardware-layer solutions for dirty tracking at sub-page granularity [9], [51] require special hardware support and is used to address specific usage scenarios such as disaggregated memory and capturing VM snapshots. The flexibility required by the OS to manage and consume dirty tracking information in a generic manner is not trivial to achieve using these hardware extensions.

Design Approach. We propose *Prosper*, a hardware-assisted checkpoint mechanism for the stack region to achieve efficient process persistence. The hardware assistance provided by *Prosper* can track stack modifications at a finer granularity with very little overhead, reducing data copy overheads associated with dirty tracking at page granularity. To provide greater flexibility to the OS, we propose a hardware-software co-design approach where the OS can control and take advantage of *Prosper* to checkpoint the stack region efficiently. Further,

the design of *Prosper* allows the OS to combine existing hardware-layer solutions for memory persistence with *Prosper* for different memory regions in the process address space.

Key results. Our experiments show that the performance overhead introduced by the *Prosper* hardware extension is, on an average less than 1% (maximum ∼3%). Leveraging dirty tracking at sub-page granularity, *Prosper* significantly reduces (on average ∼4×) the amount of data copied during checkpoint and improves the overall checkpoint time.

For a workload performing sparse writes to the stack region, *Prosper* reduces checkpoint size by 99% compared to page granularity dirty tracking, resulting in ∼22× improvement in the time taken to checkpoint the stack region. *Prosper* performs better than state-of-the-art NVM memory persistence schemes such as Romulus [15] and SSP [41] for providing stack persistence. *Prosper* provided up to 3.6× reduction in stack persistence overhead with respect to SSP and a maximum of 1.27× reduction with respect to page-level Dirtybit mechanism. A process persistence solution combining *Prosper* and SSP results in up to 2.6× improvement in achieving memory state persistence compared to a scenario when only SSP is used for the entire memory.

The summary of contributions is as follows,

- We motivate the need for specialized techniques for program stack persistence (Section II).
- We design and implement hardware extensions for efficient dirty tracking at sub-page granularity to reduce overheads associated with stack checkpoints (Section III).
- We demonstrate the seamless integration of *Prosper* with OS-layer process persistence solution along with other generic memory persistence solutions (Section III-D).
- Finally, we empirically demonstrate the efficacy of *Prosper* in terms of its dirty tracking efficiency and its positive impacts towards achieving the process memory state persistence (Section IV).

II. Motivation

For the experiments presented in this section, we traced the stack usage of some memory-intensive application benchmarks (Figure 1) using SniP [29], an open-source stack tracing framework, on a four-core Intel(R) Xeon(R) W-2104 system. For Gapbs_pr, the input parameters are: `kronecker` graph with 2^{27} vertices, 1000 iterations, $1e^{-4}$ tolerance, and 16 trials. G500_sssp uses scale as 16 and edge factor as 64. For Ycsb_mem, we traced Memcached while performing YCSB `workload-A load` followed by `workload-B run`. We traced for the highest weighted interval identified by SimPoint [23].

A. Inefficiency due to Stack Pointer Unawareness

Existing techniques without SP awareness perform non-trivial operations (e.g., create a log entry) throughout the interval to maintain the persistence state of the stack. The overhead of such operations depends upon the specific persistence mechanism under consideration. For example, a log-based scheme may create log entries for each write to the stack

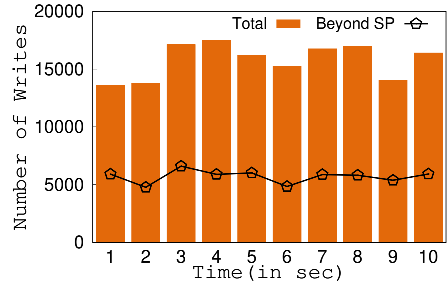
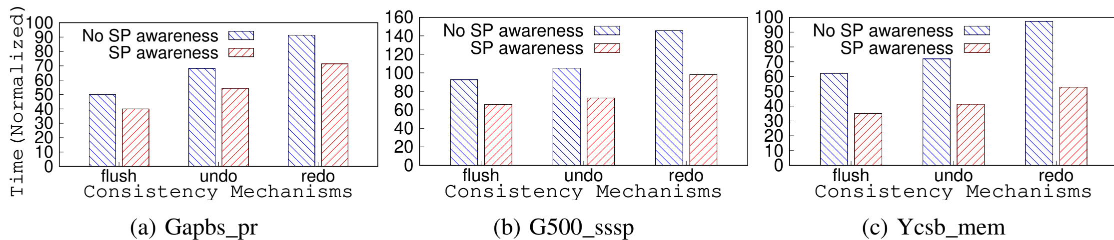


Fig. 2: Number of total stack writes and writes beyond final stack pointer (SP) aggregated at 100 intervals with each interval of 10ms for `Ycsb_mem` benchmark.

region, resulting in overall performance overhead proportional to the number of writes along with the cost of creating and serializing log entries. In this case, the overhead incurred to persist stack is not determined by the active stack region at the end of an interval; thus a log-based scheme is not SP aware. Towards capturing the overhead quantitatively, we calculate the number of stack modifications during an interval that is beyond the active stack region (i.e., beyond the value of SP) at the end of the interval using the access traces. Figure 2 shows the total number of stack writes and writes beyond the final SP, aggregated for 100 consistency intervals with each interval of 10ms duration (used for process persistence in Aurora [48]) for the Ycsb_mem benchmark. On average, more than 36% of the stack modifications are beyond the final SP for Ycsb_mem, and the behavior is similar for other benchmarks such as Gapbs_pr and G500_sssp (not shown in Figure 2). The impact of SP unawareness on a persistence mechanism can be significant considering the non-negligible proportion of operations turning out to be wasteful.

Next, we analyze the benefit of incorporating SP awareness into common memory persistence mechanisms such as $flush$, $undo$ and $redo$, to understand the extent of performance penalty these mechanisms suffer due to *SP unawareness*. We replayed the read/write accesses in the stack memory traces using a custom program on an Intel(R) Xeon(R) Gold 6226R system with NVM (Optane DCPM [54]). The custom program performed an equivalent number of reads/writes in the trace with the configured memory persistence methods (i.e., $flush$, $undo$ or $redo$) in *"No SP awareness"* scenario, whereas it applied the method only to the active stack region in *"SP awareness"* scenario. The flush technique used a `clwb` instruction after every store operation. Note that, inherently the above memory persistence mechanisms can not have SP awareness as they have to intervene and perform operations for every write to NVM. The trace-driven replay allows us to incorporate SP awareness in these techniques for analysis purposes.

Figure 3 shows the potential benefit of incorporating SP awareness in $flush$, $undo$, and $redo$ techniques to achieve stack persistence. The results show the execution time for different mechanisms with and without SP awareness (in NVM) normalized to execution time when no persistence



(a) Gapbs_pr (b) G500_sssp (c) Ycsb_mem

Fig. 3: Execution time of primitive memory persistence mechanisms with SP awareness and No SP awareness normalized to no persistence (DRAM). SP awareness: stack pointer awareness.

mechanism is used i.e., stack region is allocated in the DRAM. We observe two important performance trends from this experiment. *First*, as shown in Figure 3, all persistence mechanisms benefit from having *"SP awareness"*; the average performance improvement compared to *"No SP awareness"* scenario across all workloads is observed to be 30%, 31%, and 33% for $flush$, $undo$ and $redo$, respectively. For example, the execution time for Gapbs_pr while using $flush$ with SP awareness is 8.5 seconds, 10.6 seconds without SP awareness and 0.2 seconds with no persistence. Even though Ycsb_mem has comparatively fewer stack modifications ($\sim$15% in Figure 1), it has more number of stack modifications beyond the active stack region compared to Gapbs_pr and G500_sssp, thus benefiting more due to *"SP awareness"*. *Second*, the overhead even with SP awareness is significant—more than 35$\times$ slowdown across all benchmarks. Techniques requiring to maintain the stack in NVM and lacking the capability to merge the consistency-preserving operations incur significant overhead considering the write-intensive nature of stack operations. For example, for $flush$, every store to the stack region would result in a write-back to NVM. A checkpoint solution, apart from allowing the stack allocation in DRAM, provides enough opportunities for coalescing as the checkpoint is performed only at the end of a checkpoint interval. Moreover, checkpoint techniques are required to perform limited operations during an interval (i.e., dirty tracking) and hence, the amount of wasted work can be minimized with efficient dirty tracking.

B. Inefficiency of Page-level Dirty Tracking

The primary sources of overhead in checkpointing any memory region are,

1) Dirty tracking overhead, i.e., overhead associated with designating dirty status of memory chunks.
2) Data copy overhead, i.e., time taken to copy modified data from DRAM to NVM.

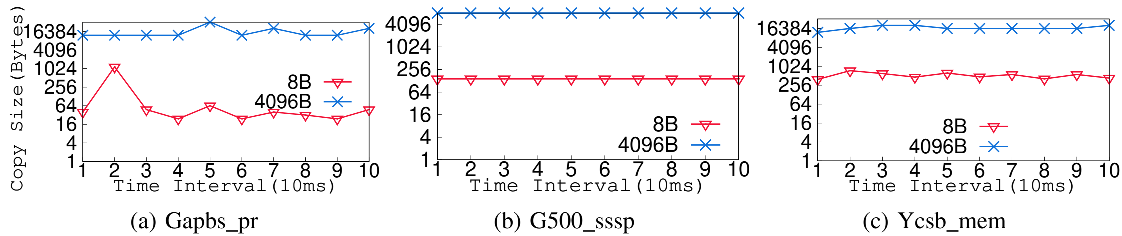
Dirty tracking techniques in contemporary systems depend upon the information gathered during virtual to physical address translation. There are two standard techniques for dirty tracking at page-level granularity.

- Using the dirty bit indicator set by the address translation hardware (e.g., the dirty bit in the page table [45]). We refer to this as the Dirtybit approach.
- Disabling the write access by write-protecting the pages in the page table [18], [49]. We refer to this approach as a write-protection based approach.

The write-protection bit-based scheme forces page faults on write access to a page during a dirty tracking interval. This scheme removes the write permission bit from the page table entries (PTEs) for all physically mapped addresses at the start of a tracking interval. Therefore, the first write to such pages in an interval would generate a page fault where the system software (OS) may record the page as dirty, which can be used at the end of the tracking period. In the Dirtybit approach, the dirty bit in the page table entries (PTEs) is reset at the start of a tracking interval. The hardware page-table walker (PTW) sets the dirty bit in PTE if there is any writes to the pages corresponding to the PTEs. At the end of the tracking interval, the OS can examine the PTEs to determine the dirty pages.

Both of the above page granularity dirty tracking techniques require the OS to walk the page table to collect dirty page information and prepare the PTEs for the next interval. However, the write-protection-based approach incurs additional overhead due to the page faults and may lead to significant overheads as shown by Singh et al. [45]. On the other hand, the Dirtybit approach is nimble and is supported by default in most of the hardware architectures. LDT [45], a technique leveraging dirty bit support of x86-64 systems, shows that the dirty tracking overhead in the Linux OS can be reduced compared to the write-protection-based technique [18]. In this paper, we use LDT [45] as the reference implementation to design Dirtybit-based approach for comparative analysis. For the stack region, dirty tracking overhead should be minimized to reduce the wasteful work during the tracking interval. However, the granularity of tracking memory modifications is limited by the address translation unit, typically an OS page [45], [49]. This can be a bottleneck in terms of *increased checkpoint size* resulting in higher copy overheads.

Ideally, a dirty tracking approach should track modifications at sub-page (or byte-level) granularity and copy only modified bytes at the end of a checkpoint interval. This conventional wisdom of tracking modifications at lower granularity [12], [41] is much more crucial for stack than other memory areas since the stack modifications majorly happen at lower granularity due to procedure calls or local variable writes. To understand the extent of reduction in checkpoint copy size with dirty tracking at byte-level (sub-page) granularity for stack



(a) Gapbs_pr (b) G500_sssp (c) Ycsb_mem

Fig. 4: Comparison of checkpoint size in page and byte level dirty tracking of stack modifications. Y axis shows copy size in page (4096 Byte) and sub-page (8 Byte) granularity tracking for benchmarks in Figure 1.

modifications, we compared data copy size in byte-level dirty tracking with conventional page-level granularity.

We post-processed the traces of benchmarks in Figure 1 to calculate the data copy size with page and 8-byte granularity dirty tracking at 10ms intervals for the stack regions. Figure 4 compares the data copy size for the page (4KB) and 8-byte granularity dirty tracking for the stack regions. Dirty tracking at sub-page byte granularity for stack reduces the checkpoint size by a factor of 300× for Gapbs_pr, 56× for G500_sssp and 33× for Ycsb_mem.

Summary. Observations presented in this section form the basis of *Prosper* where we make a case for tracking stack modifications at a finer (byte) granularity to reduce the checkpoint size. Apart from dirty tracking at a finer granularity, the proposed system by virtue of its design should allow stack allocation in DRAM, better symbiosis with the OS-layer process persistence mechanisms, support efficient software implementation to capture stack checkpoints, and limit the penalties of SP unawareness by efficient dirty tracking.

III. Dirty Tracking with Prosper

To achieve process persistence through OS-level periodic checkpoints of different process states, the memory state of the process needs to be persisted in a crash-consistent manner. For stack persistence, hardware-only approaches face non-trivial challenges due to SP unawareness and integration difficulties with the OS layer checkpoint procedures. As summarized previously, a periodic checkpoint approach for stack persistence has many advantages. However, tracking stack modifications at sub-page granularity to reduce the checkpoint size requires additional hardware support. A desirable solution should provide low-overhead dirty tracking of the stack region while supporting the OS-layer checkpointing in an organic manner. One possible design choice can be an OS-hardware co-design where the following non-trivial design challenges are addressed.

i) Separation of responsibilities along with an efficient communication protocol between the hardware and software (OS) components is necessary. OS should notify hardware to start/stop tracking at the beginning/end of any checkpoint interval. For correctness, synchronization between the OS and hardware to ensure quiescence of the dirty information before consuming it from the OS must be ensured.

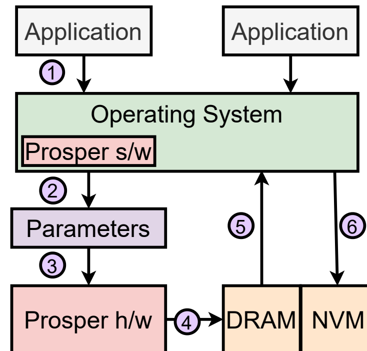


Fig. 5: Schematic diagram of Prosper.

ii) Hardware tracker should not stall load/store requests from the processor to the stack memory. Tracking must be done out of the critical path of demand requests from the processor. Hardware tracker should also generate minimum memory requests to reduce its footprint in the memory hierarchy.

iii) The OS and hardware components should co-ordinate sharing the information regarding the tracking granularity, address ranges of stacks used by different execution entities (such as threads), their corresponding meta-data regions to record/consume tracking information in an efficient manner. Across different events, such as checkpoints, context switches, etc., correctness and efficiency should be ensured.

Prosper uses a hardware-software (OS) co-design approach in which the OS records stack address range and the hardware component tracks stack modifications. Even though *Prosper* is proposed for tracking stack modifications, its generic design can be leveraged to track modifications to any virtual address range. For example, we can use *Prosper* to track modifications to dynamically allocated virtual address range in the heap.

Figure 5 shows the division of responsibilities and handshakes between the hardware and the OS components in *Prosper*.

A. Prosper Software

The software (OS) component assists the hardware component by providing required information through a set of parameters, addressing the first and third challenges of communication between software and hardware. *Prosper*'s OS component

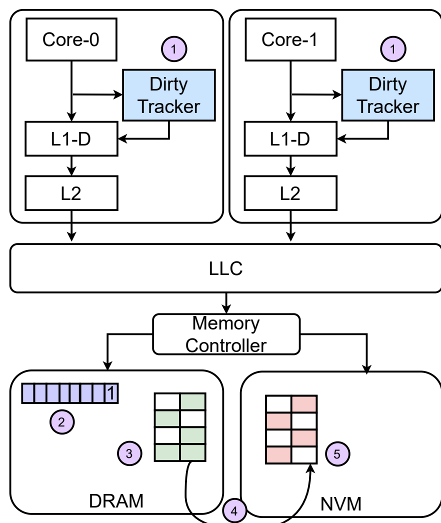


Fig. 6: Schematic diagram of Prosper hardware.

records the stack address range of an application thread (① in Figure 5) and passes it along with other information such as tracking granularity and address of memory area to record metadata about stack modifications through parameters (②). *Prosper*'s h/w component uses these parameters (③) for tracking modifications to the application's stack. *Prosper* saves the tracked dirty information in memory (④) using a bitmap, addressing the second challenge regarding metadata storage, and OS utilizes it (⑤) to decide which stack areas are modified in the current checkpoint interval. A bit in the dirty bitmap corresponds to a stack address range based on the tracking granularity. OS finally initiates a copy of data (⑥) in memory after ensuring all dirty tracking information is in a consistent state. Before performing bitmap inspection, the OS ensures quiescence of the bitmap area using a two-step process—-*(i)* instructs the *Prosper* h/w to flush all tracked dirty information, *(ii)* ensures completion of flush related activities by checking a hardware indicator. The OS may perform other activities (e.g., preparing for copy) between the two steps to reduce overheads due to stalling in the second step.

The OS clears the recorded dirty bits before starting the next checkpoint interval to ensure correct recording of dirty information in the next interval.

The efficiency of OS processing depends on performing targeted processing of the stack region where the OS examines the dirty meta-data and performs copy operations only for the active stack region. To avoid walking the entire meta-data area to clear the bits set in the last iteration, the OS should know the maximum active stack region during the checkpoint interval. The *Prosper* hardware tracks this information and shares it with the OS at the end of the checkpoint interval. The OS component also handles events such as context switches and process/thread migration, which are not shown for simplicity.

B. *Prosper Hardware*

The nucleus of *Prosper*'s hardware component is a dirty tracker hardware as shown in Figure 6. The tracker (shown as ① in Figure 6) is active during a checkpoint interval (i.e., between a checkpoint start and end). The tracker monitors memory store operations and filters the ones to the stack region without interfering with the progress of the store operation, addressing the second challenge (mentioned previously). The primary task of the tracker is to set bit(s) in the bitmap area (shown as ②) corresponding to the addresses of the filtered store(s). Each bit in the bitmap is associated with an address range in the stack based on the tracking granularity. The tracker can be configured with granularity as multiple of 8-byte.

With *Prosper*, the bitmap and volatile state of an application reside in DRAM (shown as ③ in Figure 6). A per-thread persistent stack is maintained in NVM (not shown in figure) which is consistently updated in two steps. In the first step, the OS copies (④) data to a temporary buffer in NVM (shown as ⑤ Figure 6) at the checkpoint end after inspecting the bits in the bitmap area. In the second step, the per-thread persistent stack in NVM is updated using the data copied to the temporary buffer in the first step. To reduce the overheads due to bitmap inspection and copy operations, the OS looks for coalescing opportunities within every eight bytes of the bitmap.

We build different design elements through the following series of questions related to the maintenance of the bitmap area during the tracking interval.

(i) When should the tracker issue bitmap store?

A straw-man approach could be to issue bitmap-store requests as and when a bit needs to be set in the bitmap due to stack modification. However, the straw-man strategy could interfere with the demand stores from the core because of the additional bitmap-store requests it generates. Therefore, we use a *lookup table* as a small cache within the tracker to coalesce the bitmap store requests for a given stack range. Bitmap store requests are generated due to—*(i)* eviction of an entry from the lookup table due to lack of space in the table, *(ii)* the entry has reached the coalescing threshold as explained below, *(iii)* at the end of a checkpoint interval. Each entry in the look-up table is a tuple of <*bitmap location address (64bits), bitmap value (32bits)*> (Figure 7). The lookup table has parallel search capability using the bitmap location address as the key. The target address for each bitmap store is searched in the table where a hit results in an update of bitmap value and a miss results in creating a new entry in the table, evicting an existing entry, if required. We considered two design choices while creating a new bitmap entry in the table.

1) Accumulate and Apply: Tracker creates an empty entry in the table without loading the old bitmap value from memory. Bitmap value changes are accumulated in this entry until a bitmap store request is generated for this entry. The store request is converted into a load request for the old bitmap value, then the accumulated bitmap value is merged with this

old value and stored back if required. Loading the old bitmap value is delayed until a bitmap store request is initiated. The advantage of this approach is that table entry is allocated instantaneously without waiting for completion of the load operation of the old bitmap value from memory.

2) Load and Update: Tracker issues a load request for the old bitmap value from memory into the table and updates the bitmap value in the table. The table contains the latest bitmap value when a bitmap store request is generated for this entry. The advantage of this approach is that no additional loads apart from the initial load are required when the same bitmap location is evicted from the table multiple times in an interval. The drawback is the need to delay bitmap entry allocation until the load for the old value is completed.

We use the first option, *Accumulate and Apply* in *Prosper* for creating a new bitmap entry in the table as it allows quick allocation of lookup table entries. This avoids complications of reserving an entry marked as "not ready" in the table for the duration of load and queuing of stores corresponding to the same entry.

(ii) What are the coalescing thresholds? As bits corresponding to stack modifications have coalesced in the lookup table entry, the tracker should decide on an appropriate event to issue bitmap store requests. The tracker should not be too eager or too lazy; the former may increase the interference in the memory hierarchy, while the latter can result in more evictions to accommodate new bitmap store requests. In the current design, the tracker issues a bitmap write request when the number of bits set in any lookup table entry reaches a high-water-mark (HWM) threshold. An optimal HWM attempts to strike a balance between memory bandwidth usage for bitmap store requests and the number of evictions.

(iii) What is the eviction strategy for the look-up table? As the lookup table has a limited size, the tracker should employ an eviction policy to accommodate new bitmap store requests. Under the current eviction policy, the tracker selects victims based on the number of bits set in the *bitmap value* of all table entries. The tracker evicts the entries for which the number of bits set in the look-up table is less than a threshold called low-water-mark (LWM). The tracker may evict a random entry if no suitable entries adhering to the LWM criteria are found. The rationale for this simple LWM-based design is to give priority to table entries corresponding to frequently modified stack areas. Moreover, function call and return may touch stack areas momentarily without a lot of reuse, which should be evicted from the table with higher priority.

C. Multi-threading Support

As each software thread has its own stack, the stack can be tracked on the logical CPU on which the thread is scheduled. *Prosper*'s per hardware thread dirty tracker can track the stack modifications of software threads and set bit(s) in the dedicated bitmap areas. During the process (and thread) context switch, the OS is required to save and restore the dirty tracker state (i.e., configuration and bitmap information), similar to other architectural states. During a context switch, quiescence of the bitmap area for outgoing context is ensured using the two-step process mentioned in Section III-A. Specifically, as soon as the incoming context is decided by the OS, it instructs the *Prosper* h/w to flush entries from the lookup table to update the bitmap area of the outgoing context. Next, the OS ensures completion of flushing before resuming the incoming context by loading the saved *Prosper* hardware state.

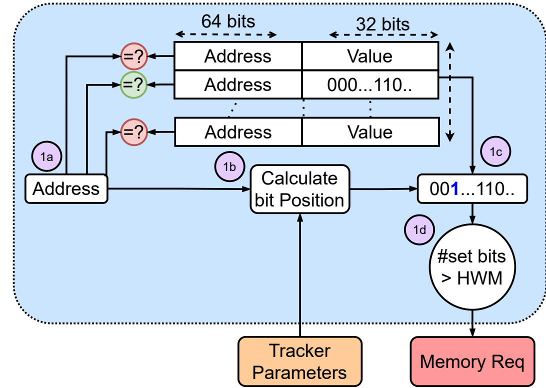


Fig. 7: Working of the Prosper hardware tracker.

One of the challenges in multi-threaded scenarios is to handle inter-thread stack modifications, i.e., when one thread of a process accesses (writes to) the stack region of another thread of the same process. This is possible because threads share the address space and can access the stack regions of each other. However, we observed that such inter-thread stack modifications are rare in applications such as the ones used in Section II. Nonetheless, the issue can be addressed by combining *Prosper* with existing privilege separation mechanisms in multi-threaded applications [53] where inter-thread stack accesses can be tracked by forcing OS interventions such as raising page faults. For *Propsper*, this can be achieved by maintaining separate page table entries for stack address ranges of different threads. The permission bits in page tables are set such that a thread has write access to its own stack but has read-only access to the stacks of other threads. On a write fault to any stack address, OS allows the write to proceed after setting the required bits in the bitmap area. Similar to the design of Wang et al. [53], changes to the stack page table of any thread need to be propagated to the page table entries of other threads in this design.

D. Implementation

The dirty tracker hardware employs mechanisms to identify and filter stores of interest (SOI). The demand store requests from the processor are inspected by the *Prosper* dirty tracker while being forwarded to the L1D. To filter the SOIs, a comparator circuit compares the store addresses against the address range set by the OS using two custom per-core model-specific registers (MSRs). Hardware dirty tracker identifies and filters required information from SOIs without impacting their progress. The comparator circuit is placed near L1D to track accesses as early as possible before they are translated or

merged. In comparison to employing tracking further down in the memory hierarchy (e.g., at memory controller), this approach has two primary benefits—*(i)* h/w filtering logic to identify SOIs becomes simple as the virtual address range for the stack is contiguous which need not be the case when filtering is based on physical addresses, *(ii)* the tracker has immediate visibility of all stack modifications which may not be possible at further levels in memory hierarchy because the accesses can be served from the upper-level cache(s).

Figure 7 shows the working of per-core dirty tracker hardware after filtering the SOIs.

The tracker uses the tracking granularity and bitmap base address values passed through two additional MSRs to calculate the corresponding bitmap address and the bit position in the *bitmap value* ((1b)). The lookup table coalesces bitmap store operations to reduce the number of bitmap store requests generated by the tracker. The tracker performs concurrent address comparison to search the bitmap location address in the lookup table (1a), by comparing against addresses stored in the lookup table. Next, if an entry exists, the tracker sets the appropriate bit in the *bitmap value* of the existing entry in the look-up table. Otherwise, it creates a new entry where only the corresponding bit for the SOI address is set in the *bitmap value* (*Accumulate and Apply* approach). The tracker finally issues a bitmap store request (1c) when the number of bits set in the lookup table value is higher than a high-water-mark (HWM) (1d). The bitmap store request for a given entry is performed in two steps—first, the old *bitmap value* is loaded by generating a load request to the address of the entry, and second, the old value is merged with value in and stored back, if necessary. The eviction operation also follows the same path, albeit the entry is marked free.

Entries in the lookup table are flushed by performing eviction of all entries when the OS indicates an end of checkpoint interval. In this case, the OS polls the tracking hardware to ensure all tracker-generated operations (load and store) are completed before proceeding further. The tracker maintains outstanding load and store request counters to ensure the completion of all in-flight operations and coordination with OS. We implement *Prosper* hardware component on gem5 [6], [35] (version 21.2.1.1).

End-to-end Checkpoint Solution A typical process checkpoint mechanism for hybrid memory systems consists of an OS layer to periodically capture the process state. Thus the OS should support hybrid memory (DRAM + NVM) and support baseline checkpoint operations for different process states. The OS on a system with *Prosper* must also incorporate additional support for *Prosper* software component. While gem5 [6], [35] supports Linux in full system mode, Linux does not support the baseline features mentioned above for a hybrid memory. Therefore, to design an end-to-end checkpoint solution using *Prosper*, we create an application checkpoint-restore infrastructure on GemOS [38], a lightweight and small OS customized for gem5 simulator. The memory management subsystem of GemOS is modified to support hybrid memory where the process uses DRAM, and stores checkpoints in the NVM (exposed through gem5). GemOS also enables periodic checkpoint operations for a process by passing the checkpoint interval as a parameter.

TABLE II: gem5 Configuration

Parameter	Used Setting	Setup
CPU	3GHz	I&II
L1-D/I	32 KiB/core (8 way, 3 cycles)	I&II
L2	512KiB/core (16 way, 12 cycles)	I&II
L3	2 MiB/core (shared) (16 way, 20 cycles)	I&II
MSHRs	16, 32, 32/core L1-D, L2, L3	I&II
Cache line size	64 B in L1, L2, L3	I&II
DRAM interface	DDR4-2400 16x4	I&II
NVM interface	PCM ‡	I
NVM Write buffer	48	I
NVM Read buffer	64	I
Memory capacity	3GB DRAM + 2GB NVM	I
Memory capacity	32GB DRAM	II
‡PCM timing parameters based on [46]		

The GemOS baseline checkpoint mechanism captures all process states (including the stack) in an incremental manner and stores them in the NVM. The memory modifications for the process are tracked at a page granularity using Dirtybit approach (Section II-B). For byte granularity checkpointing with *Prosper*, we incorporate the *Prosper* software component into GemOS to perform different handshake operations with the underlying *Prosper* hardware component using custom MSRs. To test the correctness, we emulate abrupt system crashes by killing the gem5 simulator process on the host while an application process is active within GemOS. After the crash, we restart gem5 and observe that the process within GemOS restarts from the last checkpoint successfully.

IV. Experimental Setup

We performed two sets of experiments using two setups (referred to as Setup-I and Setup-II). The first set of experiments demonstrates the end-to-end improvement of checkpoint performance with *Prosper* while the second set of experiments analyzes the hardware dirty tracking overhead introduced by *Prosper*. We used gem5 (version 21.2.1.1) [6] with configurations mentioned in Table II for the experiments. Table II lists down the NVM parameters that are different from the default NVM interface (i.e., NVM-2400 1x64) in gem5. The parameters not mentioned in Table II are set to the default settings of the gem5 simulator. Unless explicitly mentioned, we use the lookup table size as 16, HWM as 24, LWM as 8, and tracking granularity as 8 bytes for all experiments (refer to Section III-D).

A. Checkpoint Performance

We used Setup-I to demonstrate the efficacy of *Prosper* through the following experiments.

1) Performance of *Prosper* to persist the stack in consistent manner vis-a-vis other memory persistence mechanisms such as Romulus [15], SSP [41] and page-granularity checkpoint using hardware dirty bit support [45] (referred to as Dirtybit).

2) Comparatively analyze the performance of achieving process memory state persistence by combining different stack persistence techniques with SSP.
3) Performance of *Prosper* for different stack usage patterns using micro-benchmarks (Table III).

We modified GemOS [38] for this set of experiments, running on gem5 with DRAM + NVM hybrid memory. Further, we implemented Romulus and SSP in GemOS.

Romulus [15] provides memory persistence by maintaining twin copies of data, both maintained in NVM, with one copy considered backup and the other as main. The authors have proposed Romulus as a user-space library; however, since the compiler manages the stack operations, we have implemented Romulus as a hardware-software co-design to interpose stack modifications. The hardware component logs the address and size of stack modifications. The software component copies modifications from the main to backup by inspecting the log entries created by the hardware.

SSP [41] ensures memory persistence at cache line granularity using a subpage shadow paging scheme. SSP maintains two physical pages for each virtual page and distributes modifications across these two pages at cache line granularity. SSP consolidates two physical pages associated with an inactive virtual page using an OS thread. We have varied the OS page consolidation thread invocation frequency with 10 μs, 100 μs, and 1ms in the experiments (OS thread invocation frequency is not mentioned in the paper). At the end of each consistency interval, SSP writes back modified cache lines using *clwb*, sends updated bitmap in extended TLB to the SSP cache, and applies it on the commit bitmap maintained in NVM.

To study the performance of *Prosper* with different stack usage scenarios, we compare it against the page-level dirty-bit mechanism (Dirtybit) applied for the stack. The micro-benchmarks in Table III capture different stack access categories by operating on an array allocated in the function scope. The *Sparse* micro-benchmark dirties four bytes of each memory page used for stack across recursive invocation of a function. The *Random* micro-benchmark writes to a fixed number of random words while the *Stream* micro-benchmark writes to the entire stack region. The *Sparse*, *Random*, and *Stream* micro-benchmarks are designed to explore the best, average, and worst case performance of *Prosper*, respectively.

The *Quicksort* and *Recursive* micro-benchmarks capture the stack access pattern with repeated function calls and returns. Finally, we use *Normal* and *Poisson* micro-benchmarks to study the performance when the number of stack accesses between two computation code fragments follows a probability distribution. To introduce stochastic behavior in terms of the number of accesses between two compute code blocks, the number of stack writes is chosen from a normal distribution (with $\mu = 63$ and $\sigma = 20$) for the *Normal* workload. For *Poison* workload, the number writes to stack between two compute code blocks are chosen from a Poison distribution with $\lambda = 63$. The compute code block in these workloads increments a register value one thousand times.

TABLE III: Micro-benchmarks

Category	Name	Description
Access Pattern	Random	Write to random elements of an array allocated in the stack
	Stream	Write to all elements of an array allocated on stack sequentially
	Sparse	Write to 4KB spaced elements of an array allocated on the stack
Function Invocation	Quicksort	Sorting elements of an array allocated in the heap
	Recursive	Recursive function invocation with parameterized call depth
Access Intensity	Normal	Normally distributed stack writes between computation operations
	Poisson	Poisson distributed stack writes between computation operations

B. Tracking Overhead Experiments

We used gem5 with Setup-II configurations and Linux (kernel version 5.2.3) for measuring the dirty tracking overhead of the *Prosper* hardware. We modified the Linux kernel to incorporate the system software component of *Prosper*. A kernel thread is used to coordinate with the *Prosper* hardware to control and collect dirty information for the stack region(s) in every 10ms interval. At the start of an interval, the kernel thread communicates tracking parameters and stack address range to *Prosper* hardware using custom MSRs. At the end of the interval, the thread synchronizes with *Prosper* hardware to ensure the completion of tracking activity before examining the dirty tracking meta-data.

We use SSSP from Graph500 [39], PR from GAPBS [5], SPEC CPU 2017 (SPECspeed) benchmarks [14], and micro-benchmark Stream (Table III) for this study and allowed the benchmark application to run for one minute (as warm-up time) before starting the incremental checkpoint. The kernel thread performed a total of 6000 checkpoints at 10ms intervals.

V. Experimental Evaluation

In the first set of experiments, we evaluate the performance benefits of *Propser* in providing process persistence by comparing it against state-of-the-art memory persistence mechanisms using Setup-I. Furthermore, the benefit of integrating *Prosper* with state-of-the-art memory persistence mechanisms for achieving memory state persistence (heap and stack combined) is investigated. We analyze the performance of *Prosper* for different stack usage patterns. In the second set of experiments, we evaluate the hardware overhead of *Prosper*, the sensitivity of dirty tracker to HWM and LWM thresholds using Setup-II, and the energy requirements of *Prosper* hardware.

Performance of Prosper: Figure 8 shows the performance comparison of *Prosper* with existing NVM memory persistence mechanisms—Romulus, SSP, and page-level Dirtybit scheme, when used to achieve persistence of the stack region of different applications. Figure 8 shows the execution time of different applications with one of the memory persistence mechanisms applied for the stack normalized to execution time

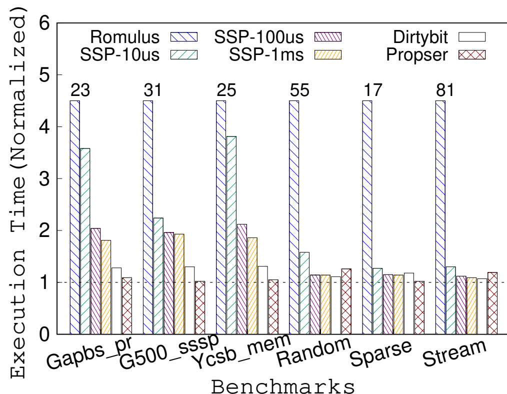


Fig. 8: Application performance comparison with different memory persistence mechanisms applied to stack. Y-axis shows application execution time with memory persistence normalized to no persistence, the lower the better.

without memory persistence. For SSP, the invocation interval for the OS page consolidation thread is varied from $10\mu s$ to $1ms$ (referred to as SSP-$10\mu s$, SSP-$100\mu s$ and SSP-$1ms$). *Propser* performs better than Romulus, SSP for all workloads and performs better than page-level Dirtybit for all except Random and Stream. SSP and Romulus require the memory area to be allocated in NVM. In contrast, Dirtybit and *Prosper* allow allocating stack in DRAM which leads to improved performance due to the access latency differences of DRAM and NVM. For SSP, the page consolidation OS thread also contributes to the performance overhead, as the merging of pages may interfere with the application execution. In Dirtybit and *Propser*, metadata inspection and data copy happen at the end of the checkpoint interval. Romulus results in significant performance overheads across all workloads as the hardware generates redo log entries for all stack modifications, and the software may copy overlapping addresses from the primary memory location to the backup memory location (in the absence of coalescing as is the case in our implementation). On the other hand, Dirtybit and *Propser* coalesce bitmap updates for the same location and avoid redundant copying at the interval end. The performance benefit of *Prosper* compared to Dirtybit results from the reduction in copy size due to the sub-page granularity dirty-tracking support of *Prosper* and efficient inspection/preparation of the dirty information metadata (Section III). *Prosper* results in an average of 2.1× (maximum of 3.6× for Ycsb_mem) reduction in stack persistence overhead compared to SSP-$10\mu s$ and a maximum of 1.27× reduction in stack persistence overhead for G500_sssp with respect to Dirtybit. The stack persistence overhead for SSP decreases with an increase in page consolidation OS thread invocation interval from 10 μs to $1ms$. For example, ∼2× reduction for Gapbs_pr from $10\mu s$ to $1ms$ is observed, but SSP incurs higher overheads compared to *Prosper* even with $1ms$ setting. *Prosper* efficiently provides stack persistence compared to other existing memory persistence mechanisms applied for stack persistence. *Prosper* design allows changing tracking granularity based on the dirty behavior of an application or disabling it to use a page-level Dirtybit scheme.

Process memory state persistence: The stack and the heap regions are two primary mutable regions in a process. To analyze the performance overhead of achieving memory state persistence of different applications, we used different combinations of SSP, Dirtybit and *Prosper* for the heap and stack segments. The combinations used for this experiment are—*(i)* SSP for both stack and heap, *(ii)* SSP for heap and Dirtybit for stack, and *(iii)* SSP for heap and *Prosper* for stack. Design of *Prosper* is inclusive enough to integrate with other memory persistence schemes such as logging for heap area.

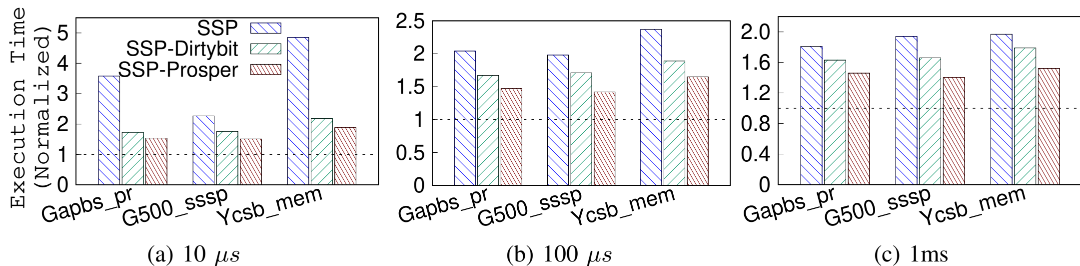
Figure 9 shows the execution time of different applications with one of the memory persistence mechanisms applied for heap and stack normalized to execution time without memory persistence. Figure 9 clearly demonstrates the benefit of combining *Prosper* or Dirtybit with SSP to achieve memory persistence compared to using SSP for the entire memory area under all three OS thread invocation intervals. SSP-Prosper performed better than SSP-Dirtybit and SSP across all three SSP page consolidation thread invocation interval scenarios. SSP-Prosper provided an average 2× (maximum of 2.6× for Ycsb_mem) reduction in memory persistence overhead compared to SSP with 10 μs thread invocation interval and an average of ∼1.4× and ∼1.3× reduction in memory persistence overhead compared to SSP with $100\mu s$ and 1ms, respectively. An increase in SSP OS thread invocation interval benefits all three stack memory persistence mechanisms, with SSP showing 2.4× reduction in memory persistence overhead for 1ms compared to $10\mu s$ for Ycsb_mem.

A combination of *Prosper* with other existing memory persistence mechanisms can provide persistence for the entire memory area with minimum overhead.

Prosper with different stack usage scenarios: We study the performance impact of different stack usage patterns with *Prosper* using micro-benchmarks in Table III. For this experiment, five different tracking granularity (8byte, 16byte, 32byte, 64byte, and 128byte) are used with a fixed checkpoint interval of 10 ms. Figure 10 shows the performance of *Prosper* for different workloads vis-a-vis the baseline, i.e., the Dirtybit (page-granularity) scheme.

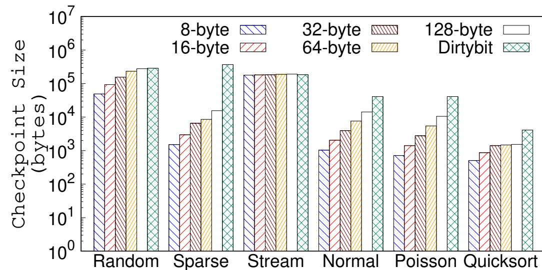
Figure 10a shows the checkpoint size for the stack averaged over all checkpoint intervals. Figure 10b shows the time taken to complete the checkpoint with *Prosper* normalized to the time taken for the page-level Dirtybit scheme. The checkpoint time consists of the time taken for inspecting dirty tracking bitmap, clearing bits in the bitmap, and copying the modifications from DRAM to NVM. While inspecting the bitmap, contiguous bits set in the bitmap are coalesced, allowing faster bitmap processing. Thus, the time for inspecting dirty tracking bitmap depends upon the bitmap area size (based on tracking granularity) and pattern of bits set in the bitmap (based on the stack access pattern of the application).

Prosper benefited the most in reducing checkpoint size when the stack modification in a checkpoint interval is lo-

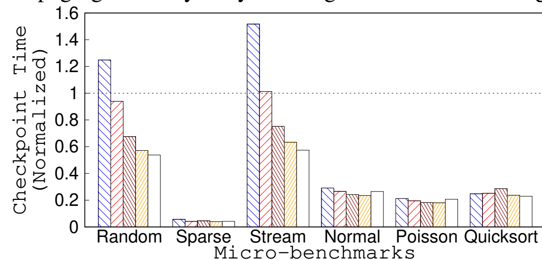


(a) 10 μs (b) 100 μs (c) 1ms

Fig. 9: Performance comparison for heap+stack area persistence, with SSP with different intervals. Y-axis shows the execution time normalized to no persistence, lower the better.



(a) Checkpoint size comparison with different tracking granularity and with page granularity dirty tracking. The Y-axis is in log scale.



(b) Checkpoint performance improvement with `Prosper`. The Y-axis shows checkpoint time with *Prosper* normalized to checkpoint time with page granularity dirty tracking.

Fig. 10: Stack checkpoint performance for different micro-benchmarks with *Prosper*.

calized to a range of stack region of size equal to or less than the tracking granularity. For example, compared to the Dirtybit scenario, checkpoint size is reduced by ∼200× for *Sparse* with 8-byte tracking granularity. Due to the significant reduction in checkpoint size for *Sparse*, maximum checkpoint time reduction is observed with all tracking granularity (average 22× compared to the baseline).

The checkpoint performance of *Prosper* is negatively impacted when the stack modification in a checkpoint interval is to a contiguous range of memory pages in the stack region (as in the case of the *Stream* workload). In such a case, checkpoint sizes for both byte-level and page-level dirty tracking are similar, and there is no reduction in the data copy overhead with *Prosper*. Apart from having no benefits of byte-granularity dirty tracking, the *Stream* benchmark incurred high checkpoint time due to the additional dirty bitmap processing operations at the end of the checkpoint and resulted in a slowdown of ∼1.5× compared to the baseline with 8-byte granularity. As the size of the dirty meta-data (i.e., dirty bitmap area in *Prosper*) decreases with increased tracking granularity, the checkpoint time overhead for *Stream* reduces with an increase in tracking granularity, showing the lowest for 128-byte granularity.

The *Random* micro-benchmark resulted in the second lowest reduction in checkpoint size compared to the page granularity checkpoint scenario. Even though 8-byte helped in reducing checkpoint size, the checkpoint time overhead of ∼1.2× in Figure 10b is due to the overhead in dirty bitmap processing. The random access pattern limits the coalescing opportunities in dirty bitmap processing. For *Random*, tracking at higher granularity helped, showing ∼1.8× reduction in checkpoint time with 128-byte, benefited from improvement in dirty bitmap processing. This is because a lower normalized checkpoint time in Figure 10b is contributed by two components: a reduction in checkpoint size with respect to the page-level Dirtybit scheme and a decrease in bitmap inspection time (decided by the bitmap area size and coalescing opportunities in bitmap). The tracking granularity plays an essential role in balancing the size of checkpoint and bitmap area size, as a higher tracking granularity reduces bitmap area size but may increase checkpoint size.

Prosper performed better with all tracking granularity for *Normal* and *Poisson* micro benchmarks. The *Quicksort* benchmark sorted elements in a heap to ensure that the stack usage is only due to function calls. Compared to the baseline, *Quicksort* performs better with *Prosper*, showing maximum checkpoint time reduction with 128-byte tracking granularity.

While *Prosper* reduces checkpoint size and checkpoint time for most stack access patterns, granularity setting should be dynamically adjusted (from the OS layer) to reduce the overhead for workloads like *Stream*.

Prosper with different checkpoint Intervals: The checkpoint interval influences the stack checkpoint size as the stack grows and shrinks multiple times during a checkpoint interval. A large checkpoint interval also allows the coalescing of multiple modifications to the same stack location in a checkpoint interval.

We studied the influence of checkpoint interval on the stack checkpoint size using function call benchmarks *Quicksort*, and *Recursive* (Table III) with call depths four (Rec-4), eight (Rec-

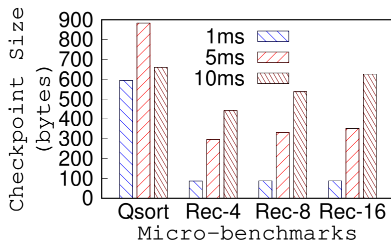


Fig. 11: Influence of checkpoint interval on checkpoint size with dirty stack tracking using Prosper.

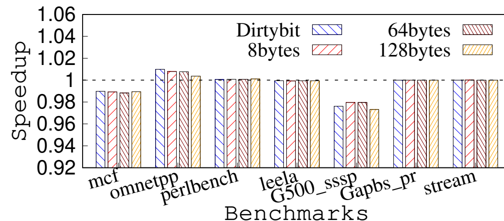


Fig. 12: Tracking overhead for different SPEC workloads. Y-axis shows an application's performance under dirty tracking with respect to no dirty tracking.

8), and sixteen (Rec-16). We used eight bytes as tracking granularity for this experiment. Figure 11 shows the checkpoint size (averaged over all checkpoint intervals) for 1 ms, 5 ms, and 10 ms checkpoint intervals. For *Recursive*, checkpoint size increases with an increase in checkpoint interval, denoting that the stack access pattern of *Recursive* does not provide coalescing opportunity, and also the stack does not shrink within the increased checkpoint interval. Whereas, stack access pattern of *Quicksort* provides benefit with increased checkpoint interval as the checkpoint size for *Quicksort* is reduced with a 10 ms interval.

We also observed that even though the checkpoint size is minimum with a 1 ms interval for *Recursive*, per byte checkpoint time (i.e., time to checkpoint a byte, measured as checkpoint time to size ratio) is the highest for *Recursive* benchmark with 1 ms interval; 22 ns with 1 ms interval in comparison to 11 ns with 10 ms interval for *Rec-4*. This is because 1 ms interval results in several checkpoints with no stack modifications (i.e., checkpoint size is 0) and incurs only dirty bitmap inspection overhead without any data copying. Therefore, very small checkpoint intervals may be counterproductive because of unnecessary bitmap inspections.

The benefit of a longer checkpoint interval depends on the stack access pattern, and having a shorter interval may be counterproductive, resulting in high checkpoint overheads.

Context switch overhead of Prosper: To study the context switch overheads, we use the Setup-I (Table II) i.e., GemOS with *Prosper* modifications executing on gem5 with *Prosper* hardware. In GemOS, while handling the timer interrupt, if the outgoing process is persistent, the OS instructs the *Prosper* hardware to flush tracked information in the lookup table to memory. The scheduling logic in GemOS continues with other activities related to context-switch, such as selecting and preparing the new context. Before scheduling the incoming context, OS ensures quiescence of the dirty tracker state for the outgoing process by checking a counter maintained in the *Prosper* hardware (Section III-C). Depending on the persistence requirement of the incoming process, OS loads the required *Prosper* parameters of the incoming context (by setting the MSRs) to notify the *Prosper* hardware.

To evaluate the context switch overhead introduced by *Prosper*, we used a multi-threaded micro-benchmark with two threads. Each thread performs a fixed number of random writes to its stack, and the main thread waits for its completion. The stack area of each thread other than the main thread is persistently maintained using *Prosper*. To measure the overhead introduced due to *Prosper* during context-switch, we capture the time taken to flush the outgoing context's dirty tracker state and set incoming context parameters. The additional overhead introduced due to the save-restore of the tracker state was observed to be ∼870 cycles on average.

Dirty Tracking Overhead of Prosper: To analyze the overheads introduced by *Prosper* due to hardware tracking of stack modifications, we performed experiments with selected benchmarks 605.mcf_s, 620.omnetpp_s, 600.perlbench_s, 641.leela_s from the SPEC CPU 2017, SSSP from Graph500, PR from GAPBS and micro-benchmark Stream (Table III).

For this experiment, we use the gem5 configuration for Setup-II (Table II) and the modified Linux kernel (see Section IV). Each benchmark application was executed initially for one minute without dirty tracking (for warm-up), and then the kernel thread performed 6000 checkpoints, each at a 10 ms interval. We used three tracking granularity—8bytes, 64bytes, and 128bytes for *Prosper*. At the end of an interval, an inspection of the bitmap area corresponding to the active stack region is performed for *Prosper*, and for Dirtybit, an inspection of dirty-bit in page table entries for the stack address range is performed.

Figure 12 shows the application's performance with dirty tracking for a fixed interval of 6000 checkpoints, calculated with respect to its performance with no dirty tracking. To isolate the performance of the benchmark application from kernel interference, we captured the number of instructions and number of cycles spent only in the user space, and speedup in Figure 12 is based on IPC in the user space. *Prosper* resulted in minimum overhead (on an average less than 1%, maximum ∼3% for G500_sssp) across all applications for all tracking granularity. Note that, even the IPC values for user mode can be impacted by the execution of OS background services (e.g., due to cache pollution), and therefore, the results should be interpreted considering the inherent variations [2], [22].

Dirty Tracker Sensitivity to HWM and LWM Parameters: The HWM and LWM parameters influence the state of the lookup table, and impact the amount of memory load and store operations to maintain the dirty bitmap area (Section III-B).

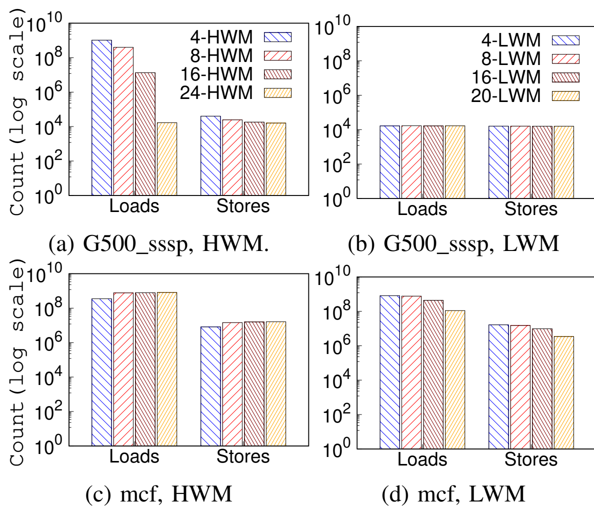


Fig. 13: Sensitivity of HWM and LWM for bitmap load and store operations. For HWM study LWM is fixed at 4 and for LWM study HWM is fixed at 24.

We analyzed the influence of HWM and LWM values on the number of bitmap loads and stores generated with mcf from SPEC CPU2017 and SSSP from Graph500 using the gem5 configuration for Setup-II (Table II) and the modified Linux kernel (see Section IV). Figure 13 shows the number of bitmap loads and stores issued by *Prosper* with varying HWM and LWM thresholds. We have fixed the LWM threshold value to 4 while varying HWM (Figure 13a and Figure 13c) and the HWM value to 24 while varying the LWM (Figure 13b and Figure 13d) for this study. For SSSP, the number of bitmap loads and stores decreases with an increase in HWM, indicating spatial locality in its stack access. At the same time, LWM variation marginally influences the loads and stores, indicating that creating more vacancies in the lookup table has no added benefits. On the other hand, for mcf, the trend is reversed where the number of bitmap loads and stores increases with increased HWM, indicating the lack of spatial locality. Further, we observe a decrease in number of load and store operations with an increase in LWM, implying more evictions can be useful. The influence of HWM and LWM on bitmap loads and stores depends on the stack access pattern. While we have used a fixed setting (LWM = 8, HWM = 24) in the previous experiments, a dynamic scheme based on the access pattern is left as a future direction.

Energy and area overhead: We obtain the dynamic energy consumption for read and write operations on the lookup table (Section III-B) configured with two read ports and one write port, using CACTI-P [33] for 7nm FINFET technology. The total dynamic read energy per access is 0.0000773194 nJ, the write energy per access is 0.000128375 nJ, and the leakage power of a bank is 0.01067596 mW. The lookup table with 16 entries occupies a cache area of 0.000704786 mm^2.

VI. Related Work

Persistent process semantics allow the resumption of an application from its last saved consistent state. In traditional OSes implementing application execution around the classical process/thread execution model, data, and pointer references to the data last till the lifespan of a process [10]. This requires special mechanisms and support from OS to persist the state of a process in a crash-consistent manner. Recent research such as NV-Process [34] provides a fault-tolerant process abstraction using NVM by decoupling the notion of processes from OS. On the other hand, Twizzler [7] proposes a data-centric OS for NVM, extending the design principles of Grasshopper [16], a single-level store OS. *Proper* is designed to handle stack persistence in an efficient manner which can be leveraged by traditional OSes as well as specialized OSes such as NV-Process and Twizzler.

Maintaining the memory state persistence in a consistent manner requires specialized techniques. The existing mechanisms for memory persistence in hybrid memory systems can be broadly categorized as *tracking-based* and *non-tracking* based on the method they use to capture memory changes.

In tracking-based techniques, memory modifications are tracked in the hardware and/or software, and persisted in a periodic manner. Aurora [48] tracks memory changes using the per-page dirty bit [21] to provide checkpoint-based process persistence in the OS layer by incrementally saving various subsystem states associated with a process, including memory. Kona [9] proposes tracking memory modifications at cache line granularity using a memory exposed through FPGA. Improvements in memory dirty tracking techniques using software and hardware enhancements ([9], [42], [45], [51]) attempt to reduce the dirty tracking overhead or support dirty tracking at a finer granularity. However, OS-driven checkpoint solutions not only require tracking at finer granularity, but also require flexibility in terms of programming/orchestrating the additional hardware support from the OS layer. *Prosper* proposes an efficient hardware tracking mechanism at sub-page byte granularity designed to integrate with the OS-layer checkpoint solutions in an organic manner.

Non-tracking techniques employ two main approaches—logging [20], [28], [50], [52], [57], [58] and shadow paging [40], [41]. SSP [41] is a shadow paging-based mechanism at cache-line granularity that redirects modifications to two different physical pages using hardware-assisted cache line remapping and consolidates these pages using a background OS thread. Atom [27] uses hardware-based undo logging and manages log allocation, ordering, and truncation in the hardware. InCLL [12] is based on in-cacheline undo logging for providing fine-grained checkpointing. Compiler-assisted techniques can add log instructions for each store inside a transaction and use special log-registers to improve the performance of log-based checkpoint solutions. Shin et al. [44] use hardware support to order log write and data update operations to realize an efficient compiler-assisted logging solution. LOC [36] uses logging by extending CPU load/store

interface and cache, ensuring memory persistence through a relaxed order of writes within and between persistent memory transactions (PTM). Capri [26] modifies the compiler to assist the hardware in maintaining undo+redo logs in a targeted fashion. HOOP [8] uses hardware-based redo logging while JUSTDO [24] is a software logging approach that minimizes log size by storing only the most recent store instruction executed within an atomic session.

ThyNVM [43] provides a hardware-assisted dual checkpointing scheme for DRAM+NVM hybrid memory that dynamically decides the checkpoint size to reduce overhead. The software, hardware, or software-hardware combined approaches mentioned above require non-trivial operation during a persistence interval. Therefore, applying them for stack persistence loses the opportunity to limit the persistence overhead to the active stack region at the end of a persistence interval. On the other hand, *Prosper* is designed to limit the persistence overhead by efficient dirty tracking at configurable tracking granularity during the interval and being stack-usage aware.

VII. Conclusion

Process persistence requires persisting its memory state consisting of mutable stack and heap segments. In this paper, we present *Prosper*, a sub-page byte granularity checkpoint based persistence mechanism for process stack that handles unique stack properties, providing an average of 2.1× (maximum of 3.6×) reduction in stack persistence overhead with respect to state-of-the-art memory persistence mechanism (SSP). We showed that *Propser* complements well with existing memory persistence mechanisms for persisting the entire memory area of a process for process persistence; *Prosper* with SSP provided an average 2× (maximum of 2.6×) reduction in memory persistence overhead for persisting the entire memory area of a process. Our evaluation using SPEC CPU 2017, SSSP from Graph500, PR from GAPBS showed that *Prosper* causes negligible tracking overhead compared to baseline (on an average less than 1%). *Prosper* addresses the unique stack properties for achieving stack persistence efficiently that can complement different varieties of existing application checkpoint mechanisms in hybrid memory systems.

Acknowledgements

We thank all the anonymous reviewers for their valuable feedback. We would like to acknowledge the efforts of Naveen MV (MTech. 2019-2021) in implementing an initial prototype as part of his MTech project. We also thank the members of the CDOS research group at the CSE department (IIT Kanpur) for their valuable feedback and support.

References

[1] A. Abulila, I. E. Hajj, M. Jung, and N. S. Kim, "Asap: architecture support for asynchronous persistence," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, 2022, pp. 306–319.

[2] A. Alameldeen and D. Wood, "Ipc considered harmful for multiprocessor workloads," *IEEE Micro*, vol. 26, no. 4, pp. 8–17, 2006.

[3] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating systems: Three easy pieces*. Arpaci-Dusseau Books LLC, 2018.

[4] K. Arun, D. Mishra, and B. Panda, "Empirical analysis of architectural primitives for nvram consistency," in *2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2021, pp. 172–181.

[5] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," *arXiv preprint arXiv:1508.03619*, 2015.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH computer architecture news*, vol. 39, no. 2, pp. 1–7, 2011.

[7] D. Bittman, P. Alvaro, P. Mehra, D. D. Long, and E. L. Miller, "Twizzler: a data-centric {OS} for non-volatile memory," in *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*, 2020, pp. 65–80.

[8] M. Cai, C. C. Coats, and J. Huang, "Hoop: Efficient hardware-assisted out-of-place update for non-volatile memory," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2020, pp. 584–596.

[9] I. Calciu, M. T. Imran, I. Puddu, S. Kashyap, H. A. Maruf, O. Mutlu, and A. Kolli, "Rethinking software runtimes for disaggregated memory," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 79–92.

[10] J. S. Chase, H. M. Levy, M. J. Feeley, and E. D. Lazowska, "Sharing and protection in a single-address-space operating system," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 4, pp. 271–307, 1994.

[11] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson, "Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 105–118, 2011.

[12] N. Cohen, D. T. Aksun, H. Avni, and J. R. Larus, "Fine-grain checkpointing with in-cache-line logging," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 441–454.

[13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*, 2010, pp. 143–154.

[14] S. P. E. Corporation, "Spec cpu 2017," https://www.spec.org/cpu2017/.

[15] A. Correia, P. Felber, and P. Ramalhete, "Romulus: Efficient algorithms for persistent transactional memory," in *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, 2018, pp. 271–282.

[16] A. Dearle, R. Di Bona, J. Farrow, F. Henskens, A. Lindström, J. Rosenberg, F. Vaughan *et al.*, "Grasshopper: An orthogonally persistent operating system," *Computing Systems*, vol. 7, no. 3, pp. 289–312, 1994.

[17] W. R. Dieter and J. E. Lumpp Jr, "User-level checkpointing for linuxthreads programs." in *USENIX Annual Technical Conference, FREENIX Track*, 2001, pp. 81–92.

[18] P. Emelyanov, "Soft-dirty ptes," https://www.kernel.org/doc/html/v5.4/admin-guide/mm/soft-dirty.html.

[19] E. R. Giles, K. Doshi, and P. Varman, "Softwrap: A lightweight framework for transactional support of storage class memory," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2015, pp. 1–14.

[20] J. Gu, Q. Yu, X. Wang, Z. Wang, B. Zang, H. Guan, and H. Chen, "Pisces: A scalable and efficient persistent transactional memory," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 913–928.

[21] P. Guide, "Intel® 64 and ia-32 architectures software developer's manual," *Volume 3B: System programming Guide, Part*, vol. 2, no. 11, pp. 0–40, 2011.

[22] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of error in full-system simulation," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 13–22.

[23] G. Hamerly, E. Perelman, J. Lau, and B. Calder, "Simpoint 3.0: Faster and more flexible program phase analysis," *Journal of Instruction Level Parallelism*, vol. 7, no. 4, pp. 1–28, 2005.

[24] J. Izraelevitz, T. Kelly, and A. Kolli, "Failure-atomic persistent memory updates via justdo logging," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, pp. 427–442, 2016.

[25] G. J. Janakiraman, J. R. Santos, D. Subhraveti, and Y. Turner, "Cruz: Application-transparent distributed checkpoint-restart on standard operating systems," in *2005 International Conference on Dependable Systems and Networks (DSN'05)*. IEEE, 2005, pp. 260–269.

[26] J. Jeong, J. Zeng, and C. Jung, "Capri: Compiler and architecture support for whole-system persistence," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, 2022, pp. 71–83.

[27] A. Joshi, V. Nagarajan, S. Viglas, and M. Cintra, "Atom: Atomic durability in non-volatile memory through hardware logging," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2017, pp. 361–372.

[28] A. Khorguani, T. Ropars, and N. De Palma, "Respct: fast checkpointing in non-volatile memory for multi-threaded applications," in *Proceedings of the Seventeenth European Conference on Computer Systems*, 2022, pp. 525–540.

[29] A. KP, S. Kumar, D. Mishra, and B. Panda, "Snip: an efficient stack tracing framework for multi-threaded programs," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 408–412.

[30] R. M. Krishnan, J. Kim, A. Mathew, X. Fu, A. Demeri, C. Min, and S. Kannan, "Durable transactional memory can scale with timestone," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 335–349.

[31] O. Laadan and J. Nieh, "Transparent checkpoint-restart of multiple processes on commodity operating systems." in *USENIX Annual Technical Conference*, 2007, pp. 323–336.

[32] D. Li, B. Reidys, J. Sun, T. Shull, J. Torrellas, and J. Huang, "Uniheap: managing persistent objects across managed runtimes for non-volatile memory," in *Proceedings of the 14th ACM International Conference on Systems and Storage*, 2021, pp. 1–12.

[33] S. Li, K. Chen, J. H. Ahn, J. B. Brockman, and N. P. Jouppi, "Cacti-p: Architecture-level modeling for sram-based structures with advanced leakage reduction techniques," in *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 2011, pp. 694–701.

[34] X. Li, K. Lu, X. Wang, and X. Zhou, "Nv-process: a fault-tolerance process model based on non-volatile memory," in *Proceedings of the Asia-Pacific Workshop on Systems*, 2012, pp. 1–6.

[35] J. Lowe-Power, A. M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj *et al.*, "The gem5 simulator: Version 20.0+," *arXiv preprint arXiv:2007.03152*, 2020.

[36] Y. Lu, J. Shu, L. Sun, and O. Mutlu, "Loose-ordering consistency for persistent memory," in *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. IEEE, 2014, pp. 216–223.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *Acm sigplan notices*, vol. 40, no. 6, pp. 190–200, 2005.

[38] D. Mishra, "Gemos: Bridging the gap between architecture and operating system in computer system education," in *Proceedings of the Workshop on Computer Architecture Education*, 2019, pp. 1–8.

[39] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[40] Y. Ni, J. Zhao, D. Bittman, and E. Miller, "Reducing {NVM} writes with optimized shadow paging," in *10th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 18)*, 2018.

[41] Y. Ni, J. Zhao, H. Litz, D. Bittman, and E. L. Miller, "Ssp: Eliminating redundant writes in failure-atomic nvrams via shadow sub-paging," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2019, pp. 836–848.

[42] Y. Ozawa and T. Shinagawa, "Exploiting sub-page write protection for vm live migration," in *International Conference on Cloud Computing (CLOUD)*. IEEE, 2021, pp. 484–490.

[43] J. Ren, J. Zhao, S. Khan, J. Choi, Y. Wu, and O. Mutiu, "Thynvm: Enabling software-transparent crash consistency in persistent memory systems," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2015, pp. 672–685.

[44] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin, "Proteus: A flexible and fast software supported hardware logging approach for nvm," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, 2017, pp. 178–190.

[45] R. Singh, K. Arun, and D. Mishra, "Ldt: Lightweight dirty tracking of memory pages for x86 systems," in *2022 IEEE 29th International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2022, pp. 85–94.

[46] S. Song, A. Das, O. Mutlu, and N. Kandasamy, "Improving phase change memory performance with data content aware access," in *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management*, 2020, pp. 30–47.

[47] R. Stoyanov and M. J. Kollingbaum, "Efficient live migration of linux containers," in *International Conference on High Performance Computing*. Springer, 2018, pp. 184–193.

[48] E. Tsalapatis, R. Hancock, T. Barnes, and A. J. Mashtizadeh, "The aurora single level store operating system," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 788–803.

[49] M. Vasavada, F. Mueller, P. H. Hargrove, and E. Roman, "Comparing different approaches for incremental checkpointing: The showdown," in *Linux Symposium*, vol. 69, 2011.

[50] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 1, pp. 91–104, 2011.

[51] D. Waddington, M. Hershcovitch, S. Sundararaman, and C. Dickey, "A case for using cache line deltas for high frequency vm snapshotting," in *Proceedings of the 13th Symposium on Cloud Computing*, 2022, pp. 526–539.

[52] H. Wan, Y. Lu, Y. Xu, and J. Shu, "Empirical study of redo and undo logging in persistent memory," in *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 2016, pp. 1–6.

[53] J. Wang, X. Xiong, and P. Liu, "Between mutual trust and mutual distrust: Practical fine-grained privilege separation in multithreaded applications," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 361–373.

[54] Z. Wang, X. Liu, J. Yang, T. Michailidis, S. Swanson, and J. Zhao, "Characterizing and modeling non-volatile memory systems," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2020, pp. 496–508.

[55] C. J. Xue, Y. Zhang, Y. Chen, G. Sun, J. J. Yang, and H. Li, "Emerging non-volatile memories: Opportunities and challenges," in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/-software codesign and system synthesis*, 2011, pp. 325–334.

[56] S. Yadalam, N. Shah, X. Yu, and M. Swift, "Asap: A speculative approach to persistence," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2022, pp. 892–907.

[57] C. Ye, Y. Xu, X. Shen, Y. Sha, X. Liao, H. Jin, and Y. Solihin, "Specpmt: Speculative logging for resolving crash consistency overhead of persistent memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 762–777.

[58] M. Zhang and Y. Hua, "Silo: Speculative hardware logging for atomic durability in persistent memory," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2023, pp. 651–663.

APPENDIX

A. Abstract

The artifact contains full source code and implementation of *Prosper* and other state-of-the-art memory persistence mechanisms—SSP and Romulus, used for comparison with *Prosper*. It includes full-system architecture simulator (gem5), operating system (gemOS/Linux) modifications, and scripts to run experiments and generate outputs. The code is suitable to be executed on Linux systems.

B. Artifact check-list (meta-information)

- **Program:** For evaluating the "Performance of Prosper" subsection under experimental evaluation section, disk images containing memory traces of benchmark applications Gapbs_pr, G500_sssp, and Ycsb_mem are provided at GitHub[1].
  For evaluating the "Dirty tracking overhead of Prosper" subsection under experimental evaluation section, a disk image containing benchmark binaries is provided on GoogleDrive URL: https://drive.google.com/file/d/1QPTfRPezp3P2YrPnOFRisFDhPBD0N3Es/view.
- **Compilation:** Scripts for compilation are included, we use GCC version 8.4.0.
- **Binary:** Linux kernel binaries are included for evaluating tracking overhead of *Prosper*. We have provided details on GitHub[1].
- **Run-time environment:** A system with Ubuntu 20.04.3, having Linux kernel 5.4.0. The system has support for Linux KVM. We provide a Docker export with all required dependencies for easy setup at GitHub[1].
- **Hardware:** Intel x86-64 with hardware virtualization support and virtualization enabled in BIOS.
- **Output:** Python scripts are provided to parse gem5 stats and generate output files. Bash scripts are provided to invoke these Python scripts and format output files for easy comparison with expected results. Plots are based on this formatted data.
- **Experiments:** Manual invocation of scripts, which launch corresponding experiments and generate outputs in designated folders.
- **How much disk space required (approximately)?:** 40–50 GB of disk after compilation.
- **How much time is needed to prepare workflow (approximately)?:** 20-30 minutes for gem5 compilation and 1-2 minutes for gemOS compilation.
- **How much time is needed to complete experiments (approximately)?:** Each experiment with gemOS under the "Performance of Prosper" subsection takes three to four hours. Each experiment of Romulus in Figure 8 takes ~20 hours to run. Each experiment with Linux under the "Dirty tracking overhead of Prosper" subsection takes approximately 15 to 20 hours.
- **Publicly available?:** Yes
- **Archived (provide DOI)?:** Zenodo[2]

C. Description

1) How to access: All the source code of *Prosper* is available at GitHub[1] and Zenodo[2].

2) Hardware dependencies: Intel x86-64 with hardware virtualization support is required for using gem5's KVM CPU, allowing fast Linux booting. We used Intel(R) Xeon(R) for running our experiments.

3) Software dependencies: A Linux system that supports building gem5. We used Ubuntu 20.04.3 with Linux kernel 5.4.0 for our experiments, and the GCC version used is 8.4.0. We used Ubuntu's "expect" package to automate interaction with the gem5 console. We provide a Docker export with all required dependencies for easy setup. The link to download Docker export and instructions to use the container are available at GitHub[1].

D. Installation

Prosper installation consists of building two components—gem5 simulator and operating system (gemOS/Linux). The GitHub[1] contains bash scripts to build the gem5 with relevant modifications and compile gemOS to produce gemOS kernels required for running on gem5. We have provided pre-built Linux kernels for measuring the dirty tracking overhead of *Prosper*. The `README` file in GitHub[1] details how to use these scripts to build/run gem5, gemOS/Linux, and generate results. We also provide Python scripts to parse and format the output files along with the expected output files.

In addition to *Prosper*, the GitHub[1] also contains implementations of other state-of-the-art memory persistence mechanisms, SSP and Romulus, used for comparison with *Prosper*.

We also provide a Docker export containing dependencies required for building gem5, gemOS. How to set up a Docker container using this export is provided at GitHub[1].

E. Experiment workflow

We provide the source code of our implementation and bash scripts (in GitHub[1] to build and execute evaluations corresponding to results under the "Performance of Prosper" subsection (Figures 8, 9, 10, 11) and results under the "Dirty tracking overhead of Prosper" subsection (Figures 12, 13). The Workflow involves invoking these scripts to generate outputs.

You can run bash scripts in parallel to reduce the overall execution time of experiments as explained in the `README` file in GitHub[1].

F. Evaluation and expected results

We provide Python scripts to parse results generated by gem5 in respective output folders. We use bash scripts to invoke these Python scripts and format output files generated by Python. We have provided "expected" results under each output folder. Please refer to the `README` file in GitHub[1] for further details.

For evaluating the "Performance of Prosper", expected results are execution time normalized to time with no persistence (i.e., vanilla). We also report checkpoint size and time to checkpoint with *Prosper* normalized to checkpoint time with Dirtybit. The Python scripts produce normalized final values used in plots.

For evaluating the "Dirty tracking overhead of Prosper", the expected results are speedup with respect to no dirty tracking and count of bitmap loads, stores with different HWM and LWM values. The Python scripts produce normalized final values used in plots.

[1] https://github.com/arunkp1986/Prosper.git
[2] 10.5281/zenodo.10123527