Prosper: Program Stack Persistence in Hybrid Memory Systems

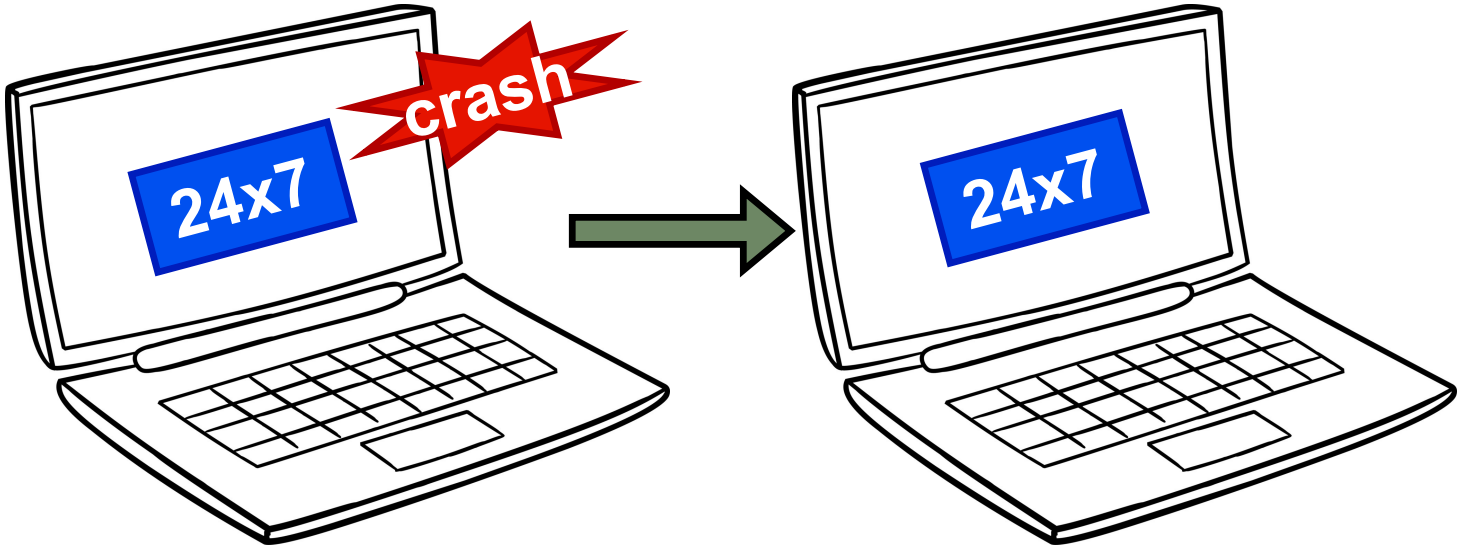
HPCA 2024

Arun KP[1] Debadatta Mishra [1] Biswabandan Panda[2]

[1] Indian Institute of Technology, Kanpur

[2] Indian Institute of Technology, Bombay

Crash Resilience



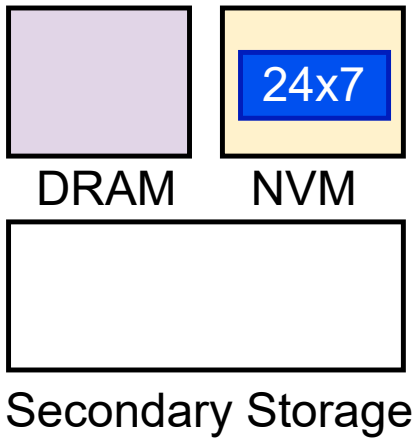
- Resume and continue execution as if crash never happened.
- Save and restore state of the process.

Persisting State Without Disk I/O

- Traditionally use storage to save state.

- Disk →file-system and storage overheads.

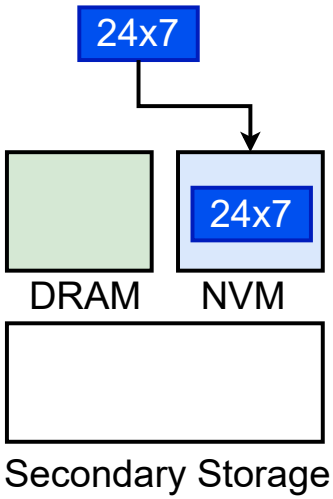
- Hybrid memory system has benefit of NVM.



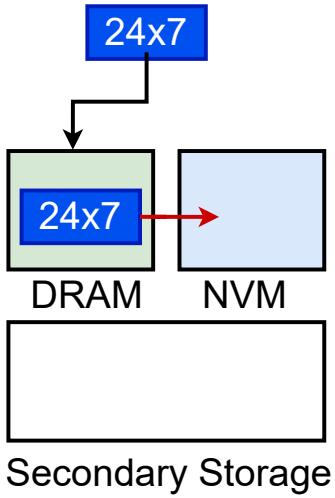
- NVM →high read/write latency.
- NVM →limited write endurance.

Maintaining State in Hybrid Memory

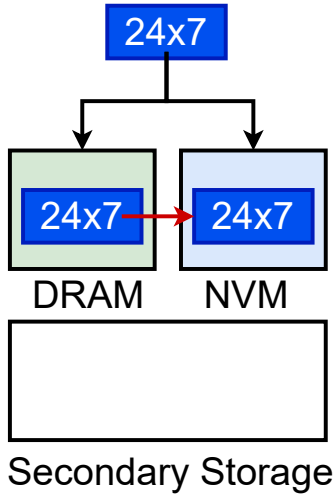
- State in NVM.



- State in DRAM and save to NVM.



- State in both DRAM/NVM and save to NVM.



Saving Process Memory State

- Memory state contributes significantly to saved state.

Saving Process Memory State

- Memory state contributes significantly to saved state.
- Efficiency of memory persistence technique is important.

Saving Process Memory State

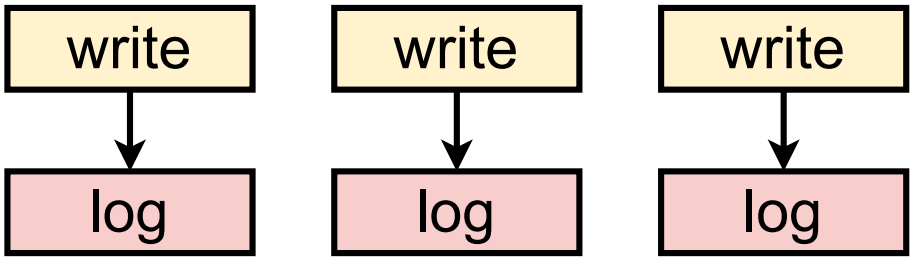
- Memory state contributes significantly to saved state.
- Efficiency of memory persistence technique is important.

Claim: Maintaining stack in DRAM is efficient in hybrid memory systems.

Techniques to persist memory state

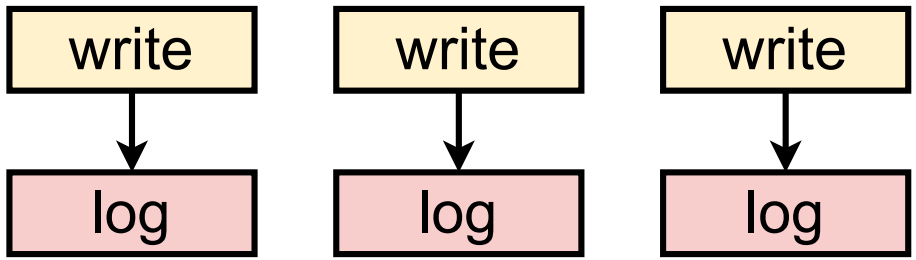
Achieving Memory Persistence

- Techniques
  - Operating in tandem with application execution.



Achieving Memory Persistence

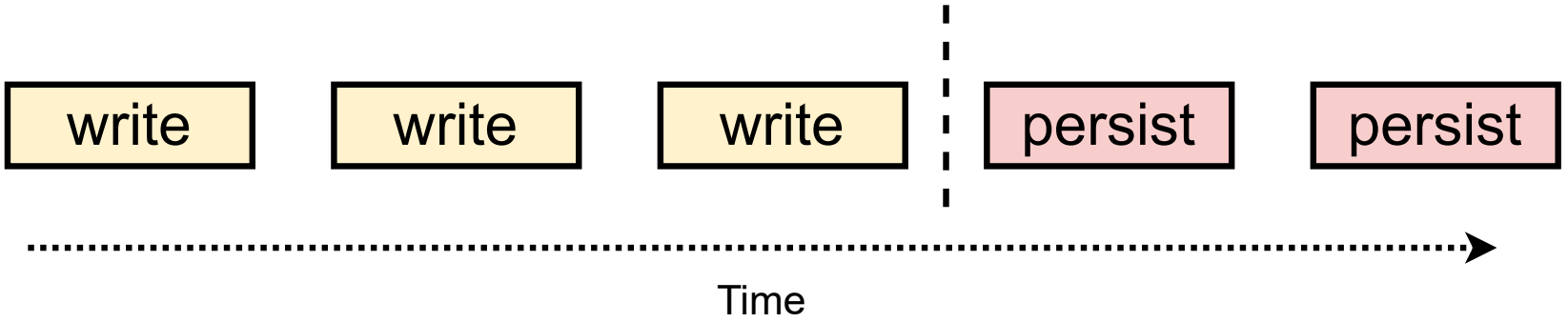
- Techniques
  - Operating in tandem with application execution.



- Timestone [ASPLOS-2020], SSP [MICRO-2019] and Romulus [SPAA-2018] are examples of techniques operating in tandem.

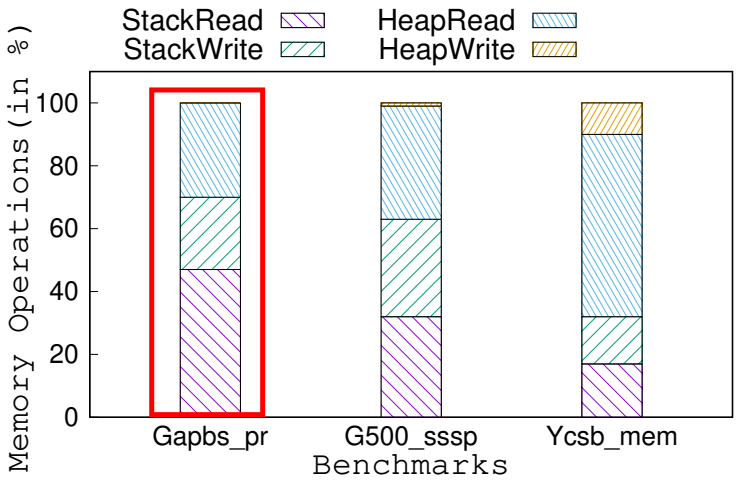
Achieving Memory Persistence

- Techniques
  - Periodic checkpointing.



Stack Size vs Operations

- Number of stack operations is significant for some applications.
- 70% of memory operations are to stack in Gapbs_pr benchmark.

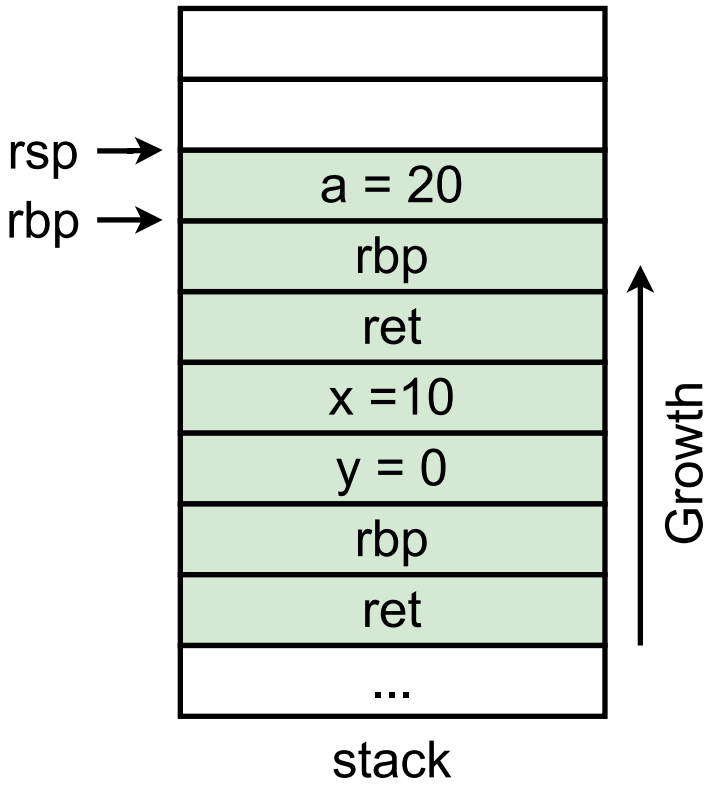


Achieving Stack Persistence

- In tandem techniques suffer due to grow-shrink pattern of stack.

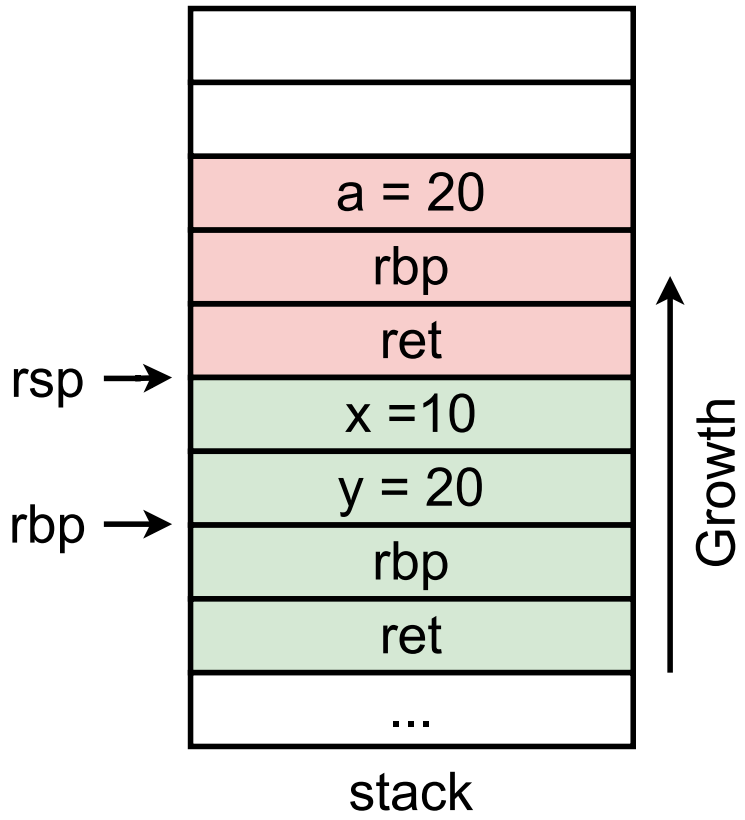
Stack Property

```
1  int doubleit(int x){
2      int a = x;
3      a = a*2;
4      return a;  <---
5  }
6  int main(){
7      int x = 10;
8      int y = 0;
9      y = doubleit(x);
10     return 0;
11 }
```



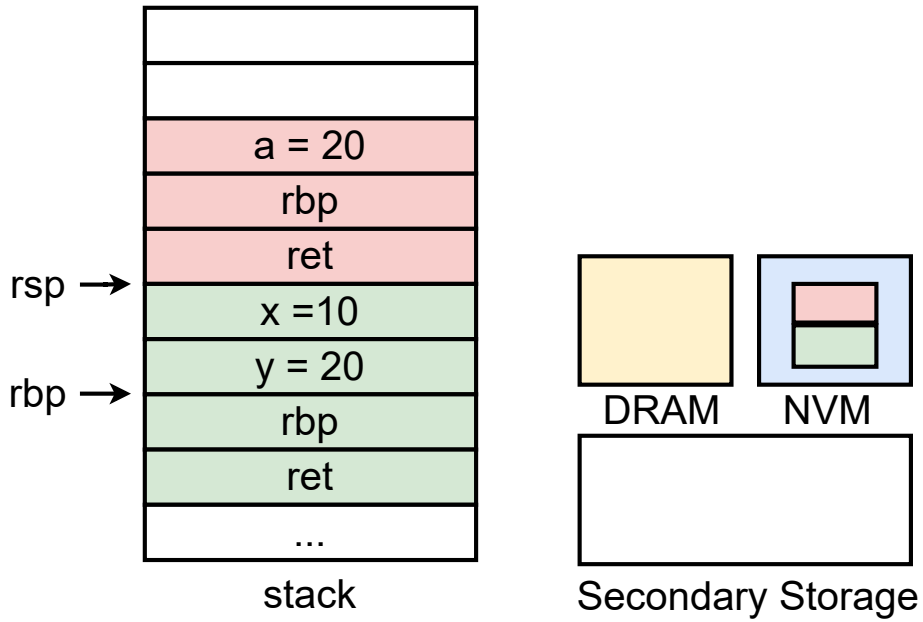
Stack Property

```
1  int doubleit(int x){
2      int a = x;
3      a = a*2;
4      return a;
5  }
6
7  int main(){
8      int x = 10;
9      int y = 0;
10     y = doubleit(x);  <---
11     return 0;
12 }
```



Achieving Stack Persistence

- In tandem techniques suffer due to grow-shrink pattern of stack.
- Create entries for entire stack region.



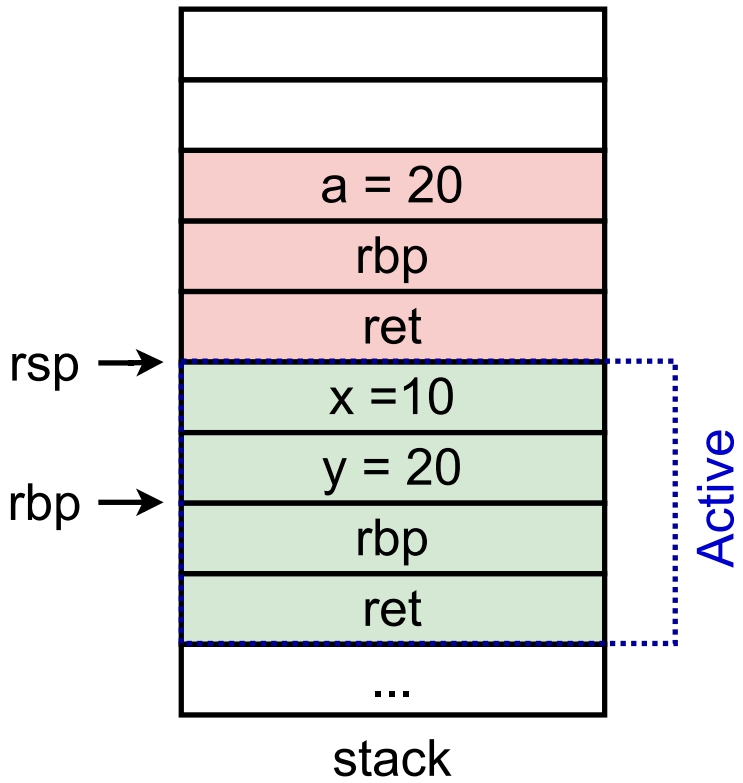
Benefits of periodic checkpointing for stack

Stack Persistence with Checkpointing

- Provides **Stack Pointer awareness** by limiting operations to active stack region.

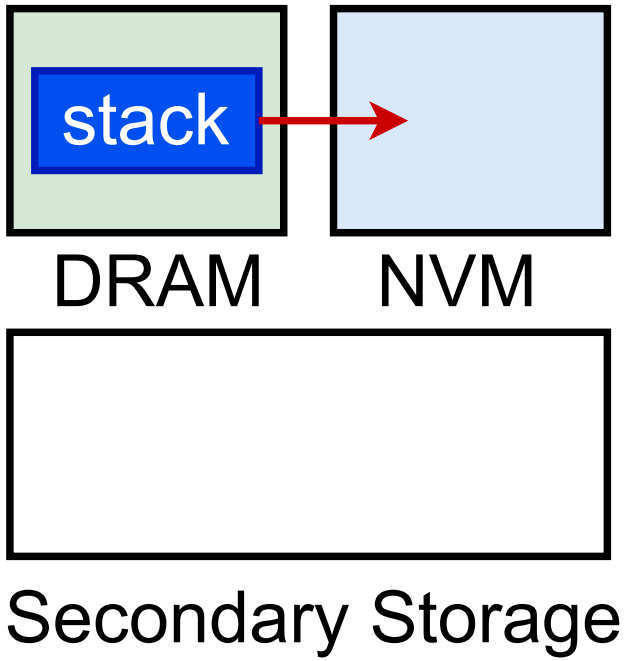
Stack Pointer Awareness

- **SP awareness** indicates knowing the active Stack Pointer at the time of persistence.



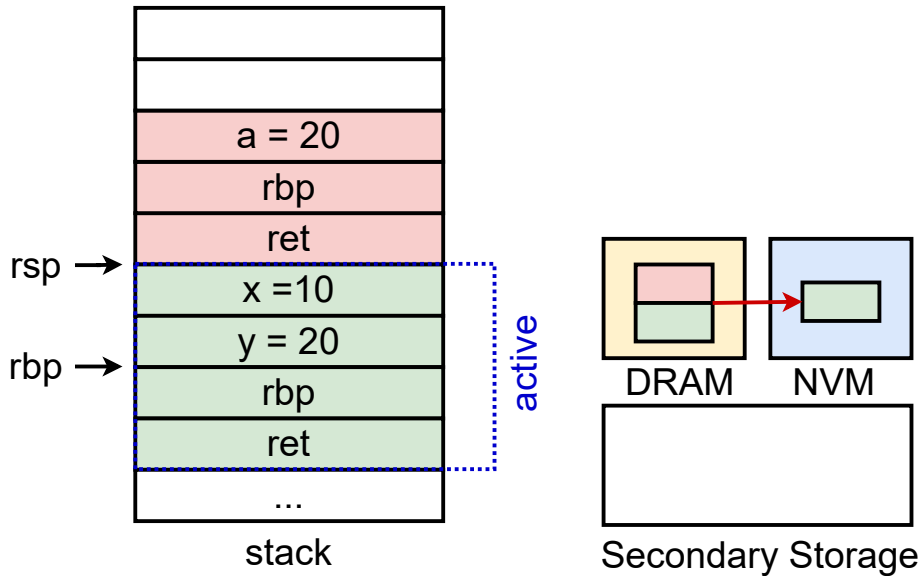
Stack Persistence with Checkpointing

- Allows maintaining stack in DRAM, **avoiding excess writes to NVM**.



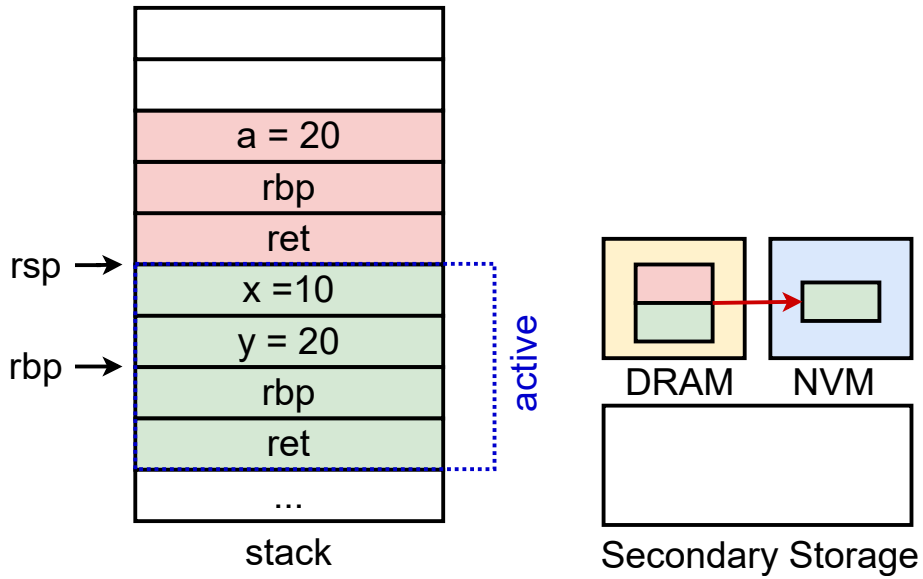
Stack Persistence with Checkpointing

- Copying from DRAM to NVM is **limited to the active stack region**.



Stack Persistence with Checkpointing

- Copying from DRAM to NVM is **limited to the active stack region**.



- Provides **visibility to stack**, handling indirect usage of stack.

Stack Persistence with Checkpointing

Periodic checkpointing is good for stack persistence.

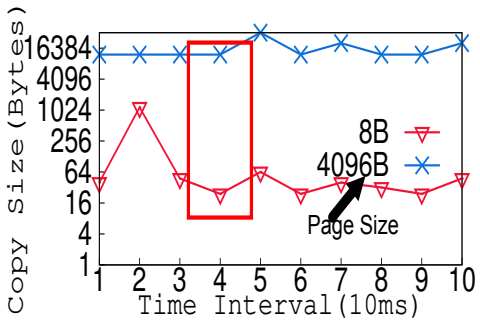
Stack Persistence with Checkpointing

Periodic checkpointing is good for stack persistence.

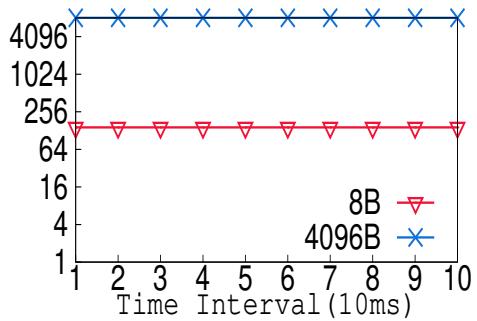
However

Checkpoint Copy Size

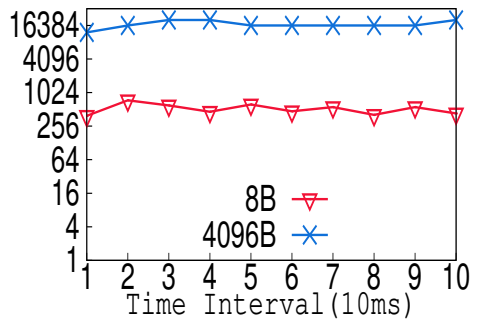
- Memory is dirty tracked at page granularity in conventional systems.



Gapbs_pr



G500_sssp



Ycsb_mem

- Sub-page granularity dirty tracking reduces checkpoint size for stack.

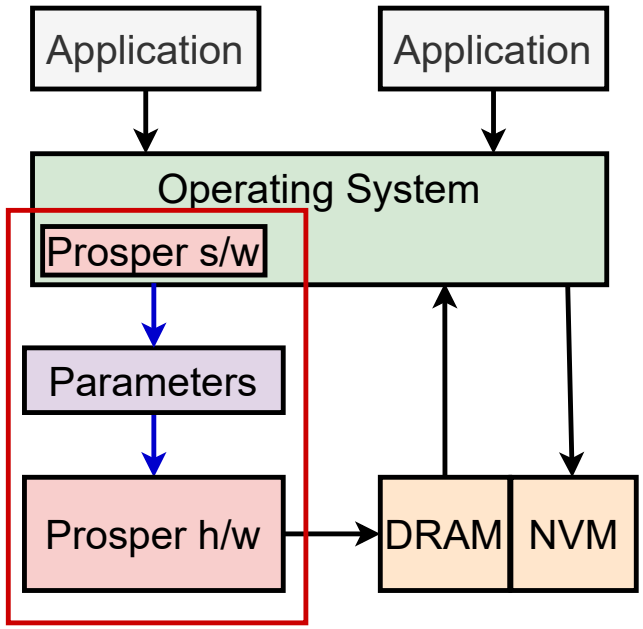
Prosper for periodic stack checkpointing

Prosper Design

- Allows checkpointing at sub-page granularity.

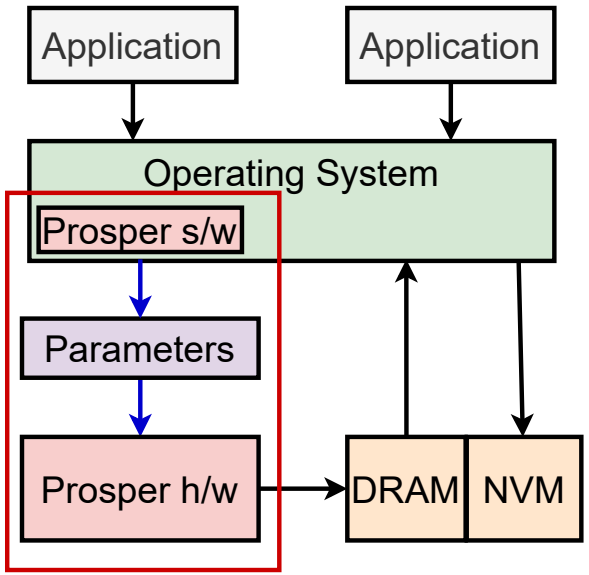
Prosper Design

- Software and hardware coordination.



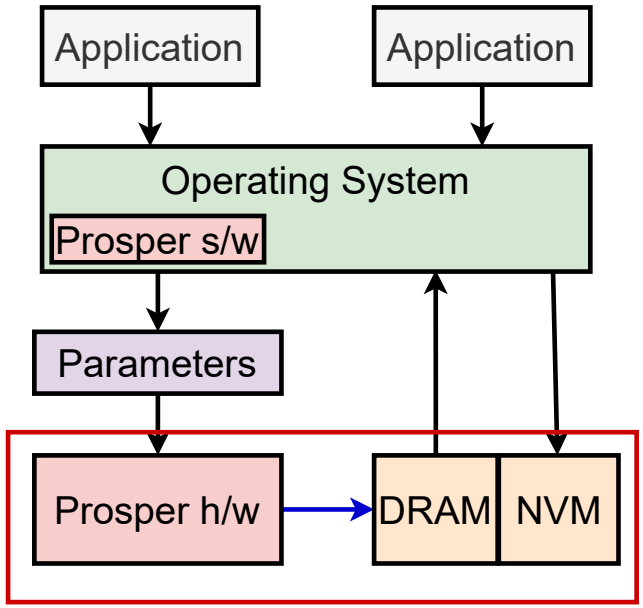
Prosper Design

- **Software component** collects and passes parameters.
- **Parameters** include stack address range, tracking granularity, bitmap area.
- **Hardware component** tracks stack modifications.



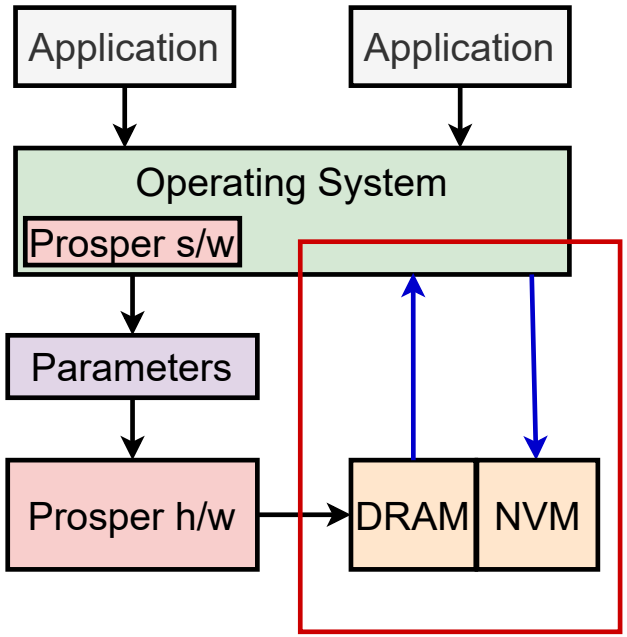
Prosper Design

- Hardware component saves bitmap in memory.



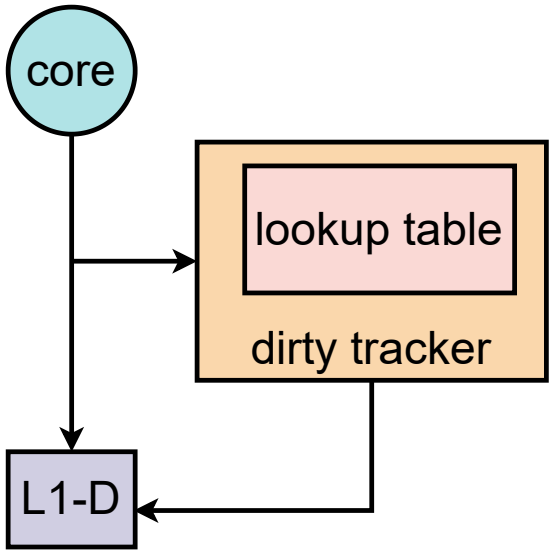
Prosper Design

- OS utilizes metadata to copy changes from DRAM to NVM.



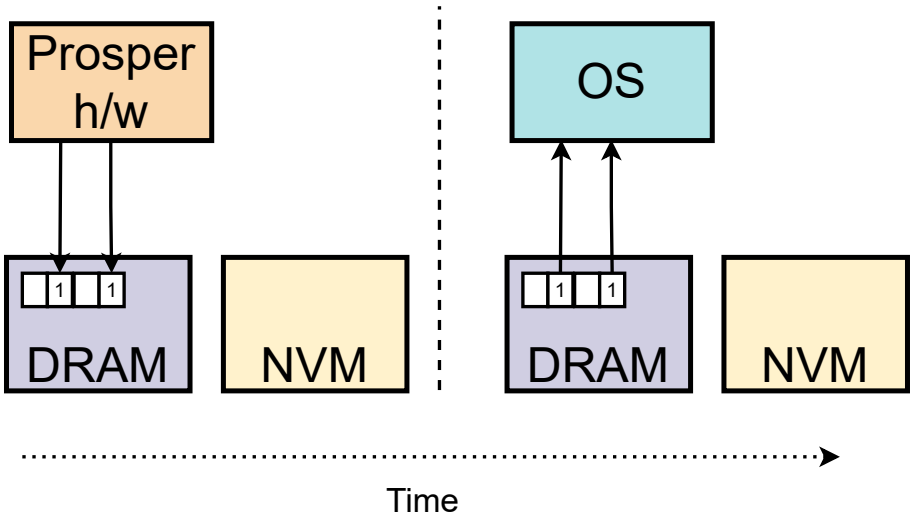
Prosper Design

- How does prosper hardware interact with loads/stores?
  - Hardware should not stall load/store requests from processor.
  - Bitmap stores should create minimum interference in memory hierarchy.



Prosper Design

- When does OS utilize meta-data?
  - OS consumes meta-data after quiescence of dirty information.



Results

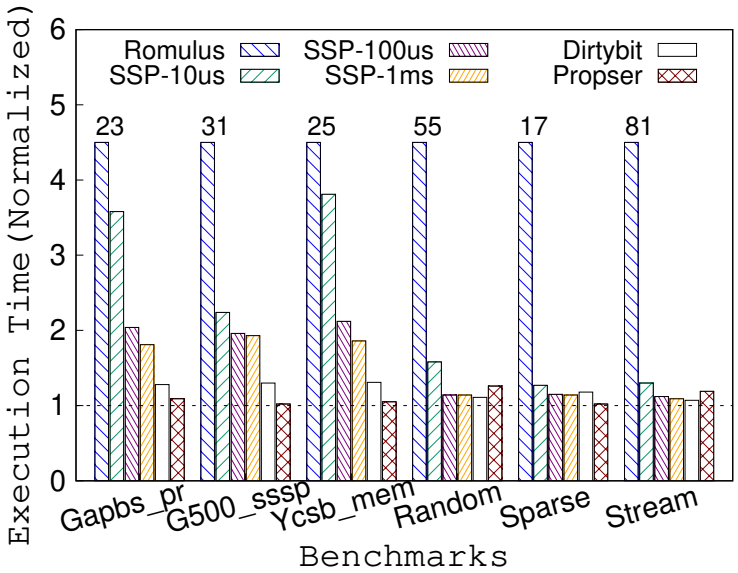
Full-System Simulation Setup

- Setup-I with GemOS, measuring end-to-end checkpoint performance.
- Setup-II with Linux, measuring hardware dirty tracking overhead.
- Table shows 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

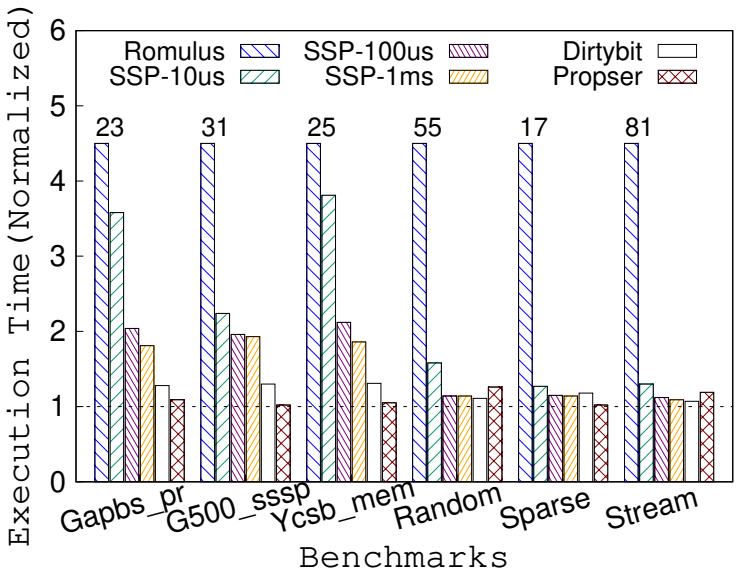
Performance of persisting stack

- **SSP thread interval varied from 10 μs to 1 ms.**
- Prosper is configured with 8 bytes tracking granularity.
- Execution time is normalized to time without memory persistence.



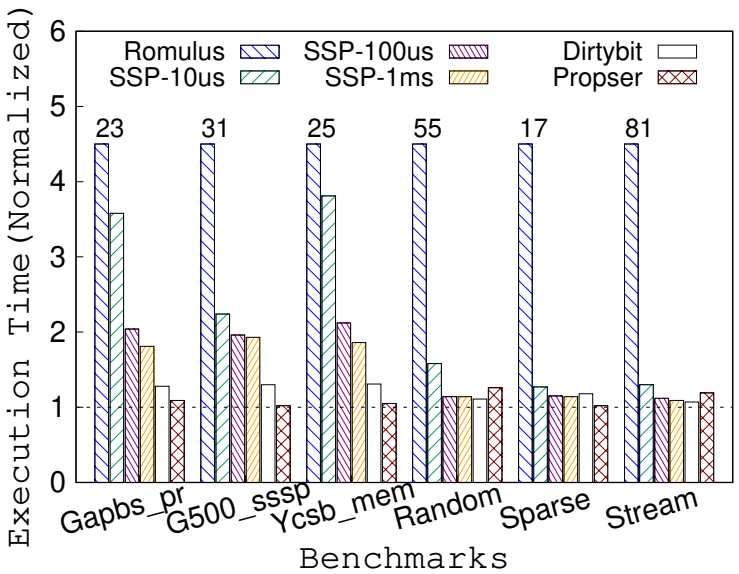
Performance of persisting stack

- SSP thread interval varied from 10 μs to 1 ms.
- **Prosper is configured with 8 bytes tracking granularity.**
- Execution time is normalized to time without memory persistence.



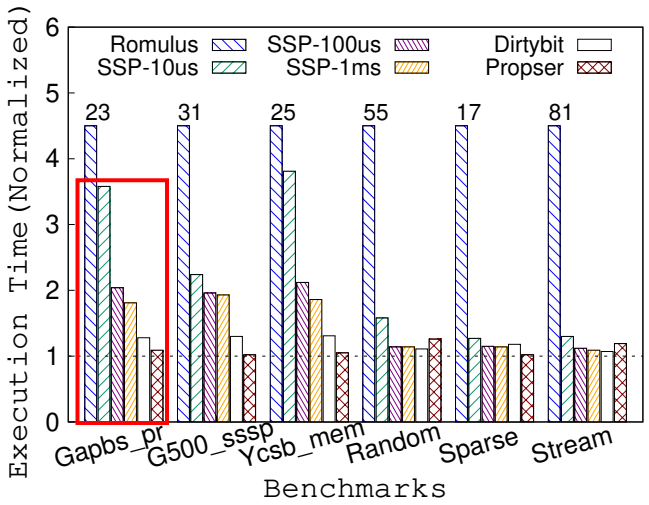
Performance of persisting stack

- SSP thread interval varied from 10 μs to 1 ms.
- Prosper is configured with 8 bytes tracking granularity.
- **Execution time is normalized to time without memory persistence.**



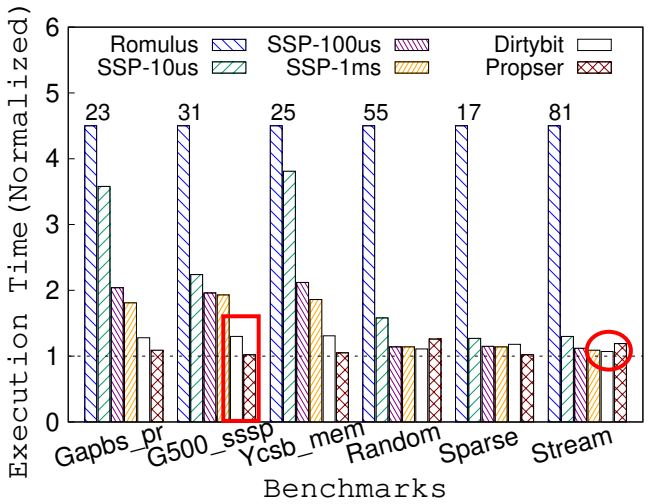
Performance of persisting stack

- Prosper reduced stack persistence overhead by 2.1× on average compared to SSP-10μs.



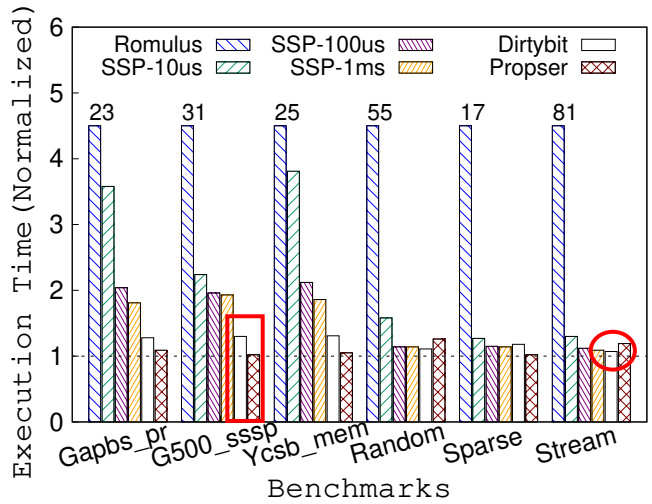
Performance of persisting stack

- Maximum of 1.27× reduction for G500_sssp with respect to Dirtybit.



Performance of persisting stack

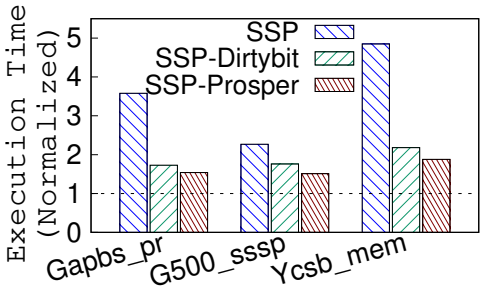
- Maximum of 1.27× reduction for G500_sssp with respect to Dirtybit.



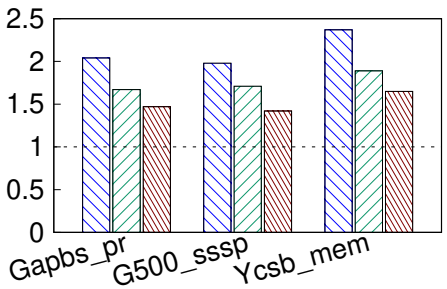
Prosper performed better than Romulus and SSP.

Performance of persisting heap+stack

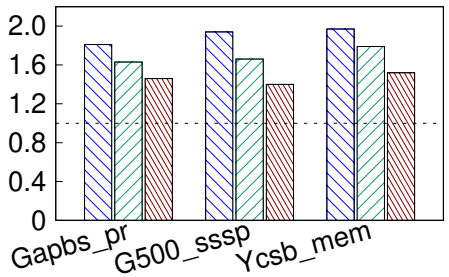
- **Combined stack persistence techniques with SSP.**
- {SSP $\rightarrow$heap+stack}, {SSP $\rightarrow$heap and Dirtybit/Prosper $\rightarrow$stack}.
- Execution time is normalized to time without memory persistence.



10 μs



100 μs

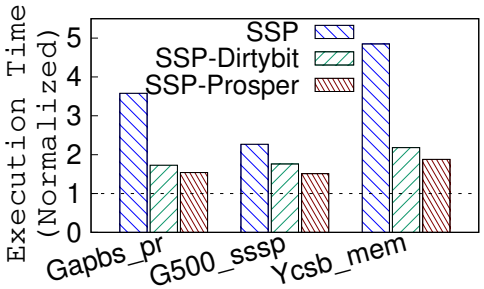


1ms

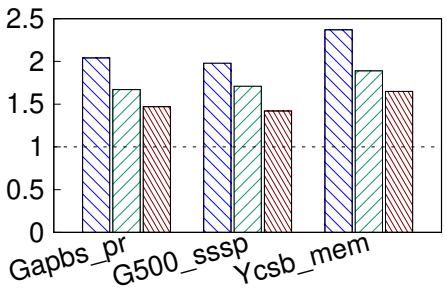
- SSP thread interval varied from 10 μs to 1 ms.

Performance of persisting heap+stack

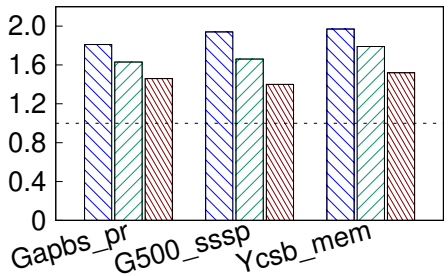
- Combined stack persistence techniques with SSP.
- **{SSP →heap+stack}, {SSP →heap and Dirtybit/Prosper →stack}.**
- Execution time is normalized to time without memory persistence.



10 μs



100 μs

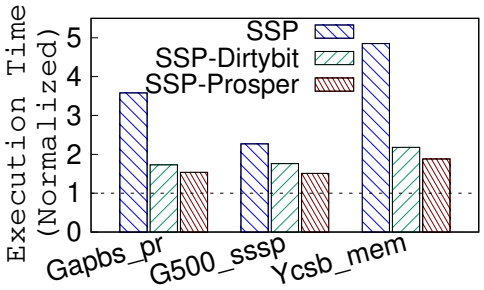


1ms

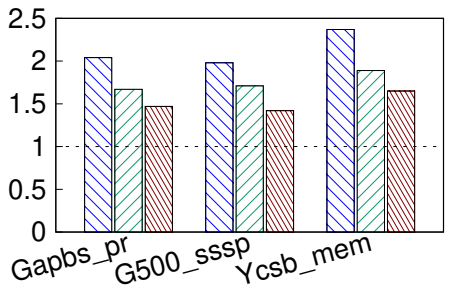
- SSP thread interval varied from 10 μs to 1 ms.

Performance of persisting heap+stack

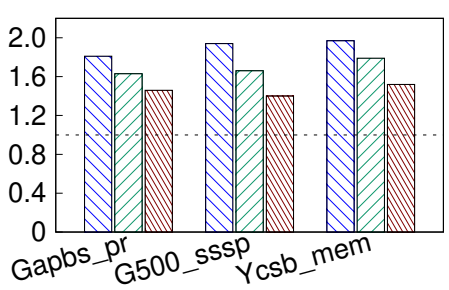
- Combined stack persistence techniques with SSP.
- {SSP $\rightarrow$heap+stack}, {SSP $\rightarrow$heap and Dirtybit/Prosper $\rightarrow$stack}.
- **Execution time is normalized to time without memory persistence.**



10 μs



100 μs

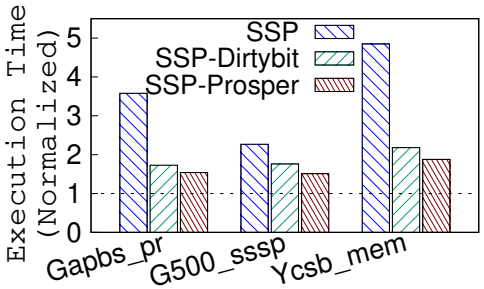


1ms

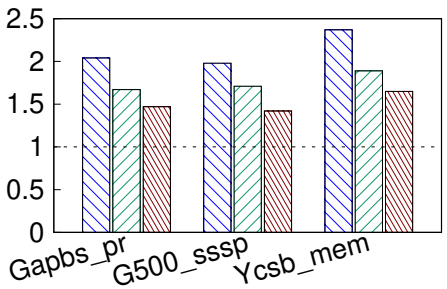
- SSP thread interval varied from 10 μs to 1 ms.

Performance of persisting heap+stack

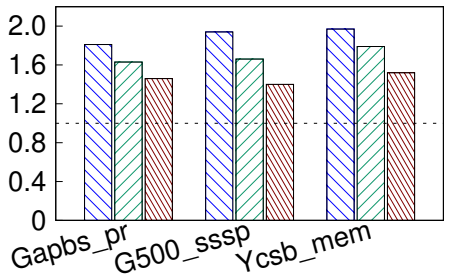
- Combined stack persistence techniques with SSP.
- {SSP $\rightarrow$heap+stack}, {SSP $\rightarrow$heap and Dirtybit/Prosper $\rightarrow$stack}.
- Execution time is normalized to time without memory persistence.



10 μs



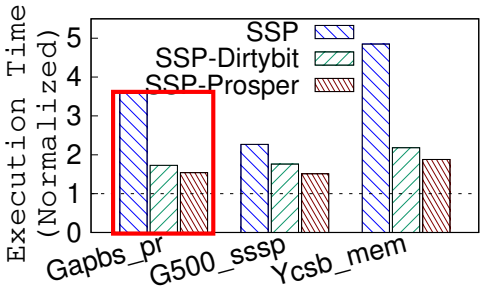
100 μs



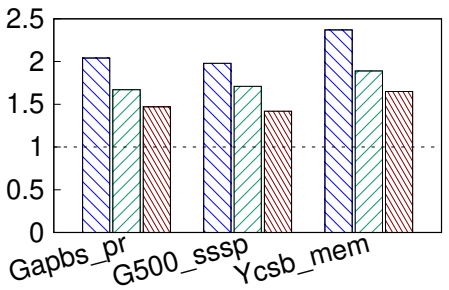
1ms

- **SSP thread interval varied from 10 μs to 1 ms.**

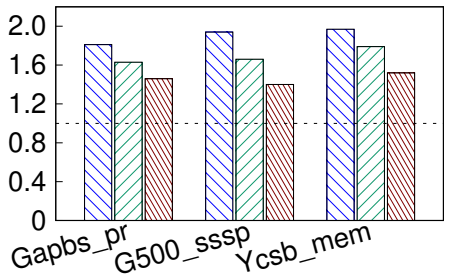
Performance of persisting heap+stack



10 μs



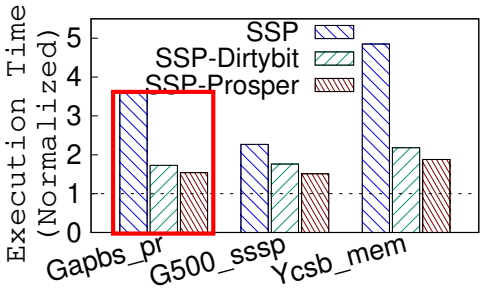
100 μs



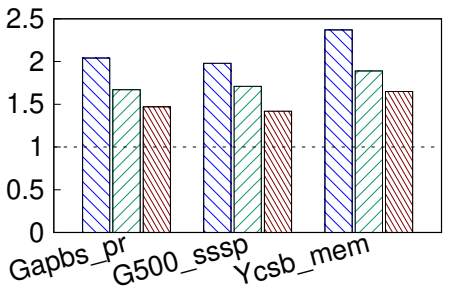
1ms

- SSP-Prosper reduced overhead by 2× on average compared to SSP with 10μs thread invocation interval.

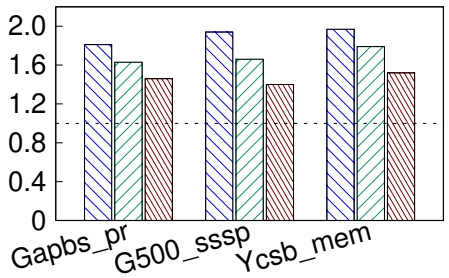
Performance of persisting heap+stack



10 μs



100 μs



1ms

- SSP-Prosper reduced overhead by 2× on average compared to SSP with 10μs thread invocation interval.

Combining Dirtybit/Prosper with SSP is better than using SSP alone.

Where Does Prosper Stand?

- Comparing with existing memory persistence mechanisms.

Property	Soft-WrAP [MSST-2015]	JUSTDO [CAN-2016]	Romulus [SPAA-2018]	SSP [MICRO-2019]	Time-stone [ASPLOS-2020]	Prosper [HPCA-2024]
Without compiler support	✗	✗	✗	✓	✗	✓
Stack pointer awareness	✗	✗	✗	✗	✗	✓
Allows stack in DRAM	✓	✗	✗	✗	✗	✓

Summary

- **Prosper provides sub-page granularity checkpointing for stack.**

Summary

- Prosper provides sub-page granularity checkpointing for stack.
- **Provides 2.1× on average (max 3.6×) reduction in stack persistence overhead compared to SSP.**

Summary

- Prosper provides sub-page granularity checkpointing for stack.
- Provides 2.1× on average (max 3.6×) reduction in stack persistence overhead compared to SSP.
- **Complements well with existing memory persistence techniques.**

Summary

- Prosper provides sub-page granularity checkpointing for stack.
- Provides 2.1× on average (max 3.6×) reduction in stack persistence overhead compared to SSP.
- Complements well with existing memory persistence techniques.
- **Prosper with SSP provides 2× on average (max 2.6×) reduction in memory persistence overhead.**

Summary

- Prosper provides sub-page granularity checkpointing for stack.
- Provides 2.1× on average (max 3.6×) reduction in stack persistence overhead compared to SSP.
- Complements well with existing memory persistence techniques.
- Prosper with SSP provides 2× on average (max 2.6×) reduction in memory persistence overhead.
- **Prosper provides stack persistence for achieving process persistence.**

Questions?

Scan for Artifact

Thank You

Scan for Artifact