

Efficient Use-After-Free Prevention with Opportunistic Page-Level Sweeping

Chanyoung Park, Hyungon Moon

NDSS 2024

25.04.03

Contents

01 Background

02 Motivation

03 Design

04 Evaluation

05 Conclusion

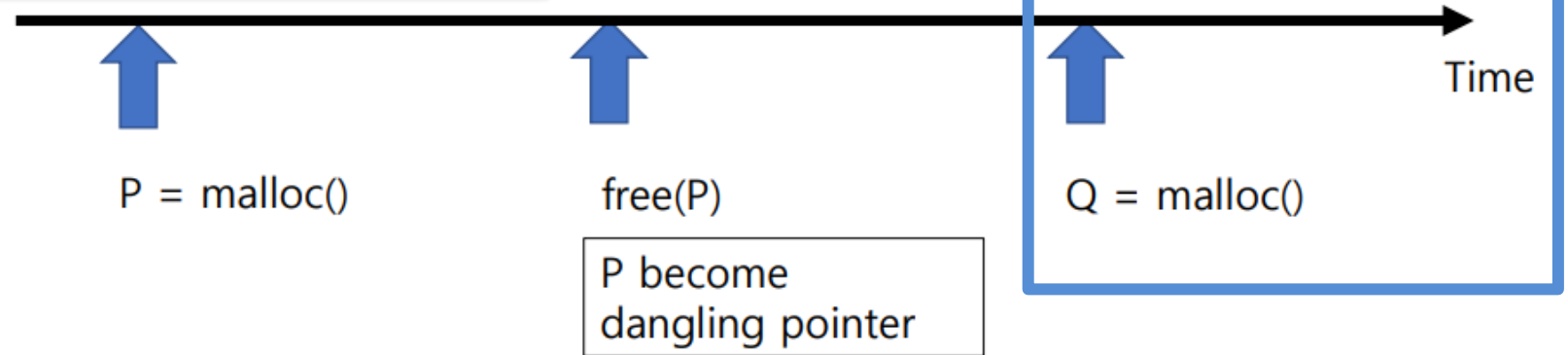
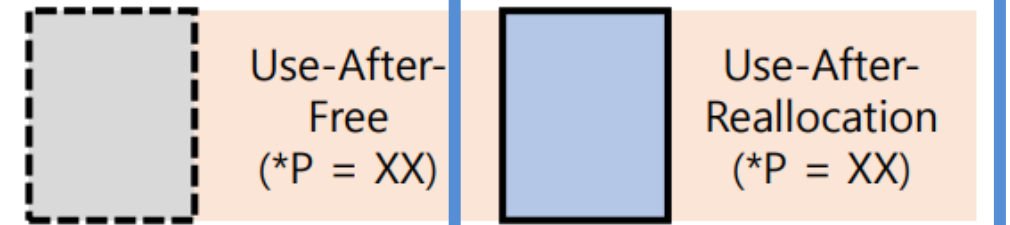
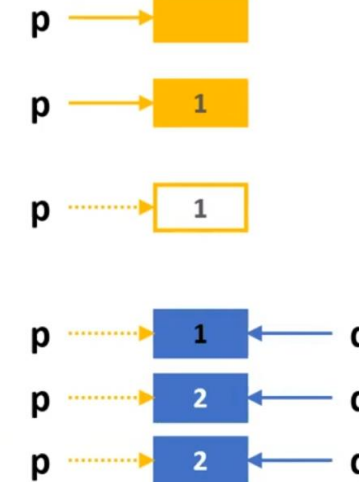


Background

Use-After-Free

- Allocation
- Use
- Free
- Re-assignment
 - Most allocators reuse p's slot for q
- Use
- Use-after-free

```
p = malloc(8);  
*p = 1;  
if (*p == 1) {  
    ...  
    free(p);  
}  
...  
q = malloc(8);  
*q = ReadNet();  
if (*p == 2)  
    ...
```



Use-After-Free Mitigation

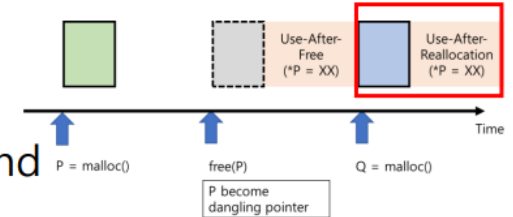
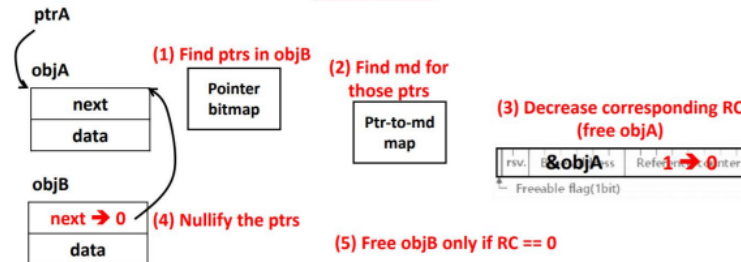
1. Pointer Invalidation
2. Delayed Reuse
3. MMU assisted Approach
4. Garbage Collection
 - MarkUs [S&P '20]
 - Minesweeper [ASPLOS '22]
5. Type-Safe Memory Reuse
6. Lock-and-Key

Garbage collection

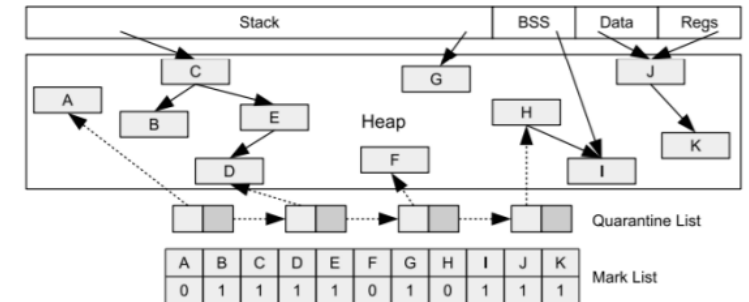
- Reclaim a chunk only if there's no reference found
 - CRCount[NDSS'19], **MarkUs[S&P'20]**

crc_free

```
4 ptrA = malloc(sizeof(struct node)); // objA
5 ptrB = malloc(sizeof(struct node)); // objB
6
7 ptrB->next = ptrA;
8
9 /* code execution */
10
11 free(ptrA);
12
13 /* code execution */
14
15 ptrA = malloc(sizeof(struct node));
16 free(ptrB);
```

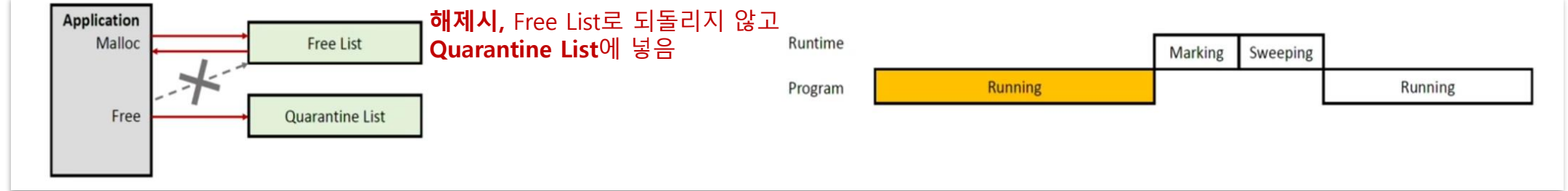


메모리를 Free했을 때, 전체 메모리를 스캔하여 해당 chunk를 가리키는 포인터가 사라졌는지 확인 후, 안전할 경우에만 재사용

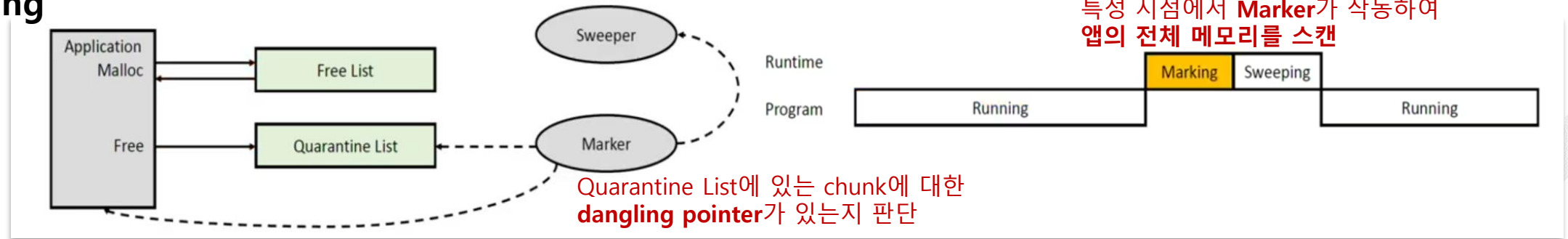


Mark-sweep Approach (Boehm GC 기반)

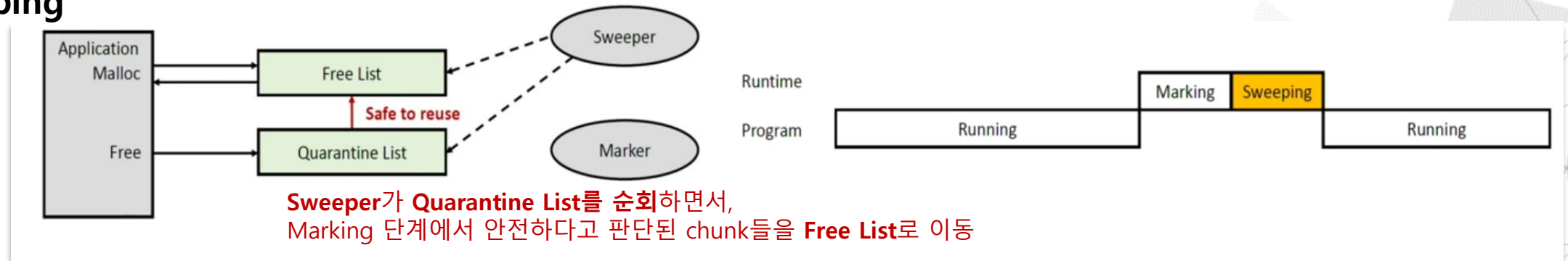
- Allocation/Free



- Marking

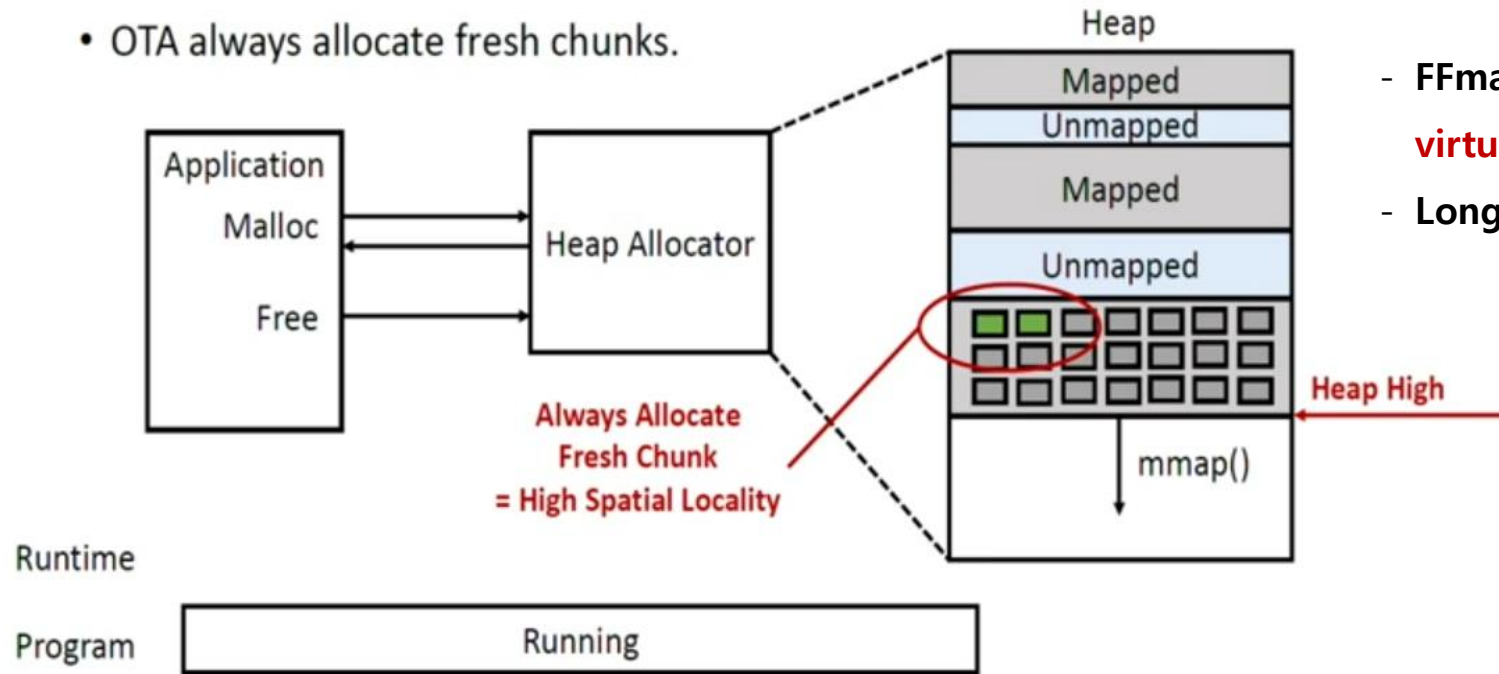


- Sweeping



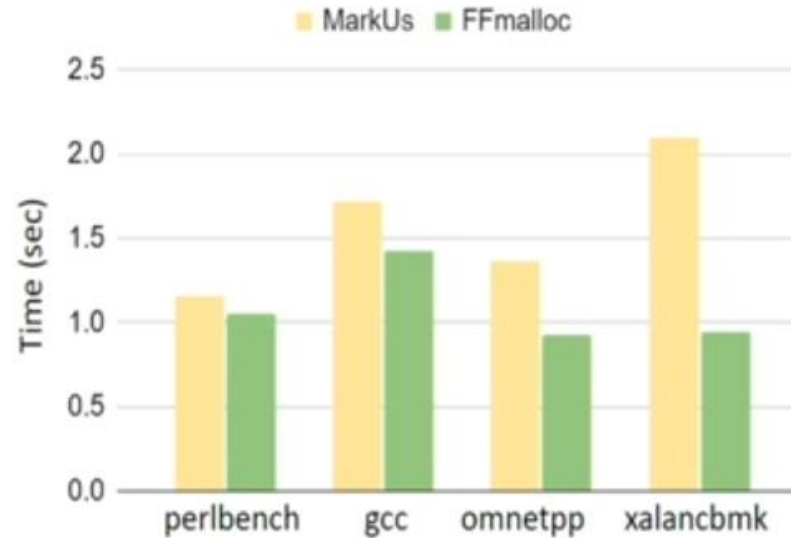
| One-time Allocation (OTA)

- OTA always allocate fresh chunks.



- FFmalloc 같은 경우에는 물리 페이지는 재사용 가능하지만, **virtual address는 계속 증가 → 결국 한계 도달**
- Long-running app엔 부적합

I MarkUs, MineSweeper의 성능 문제

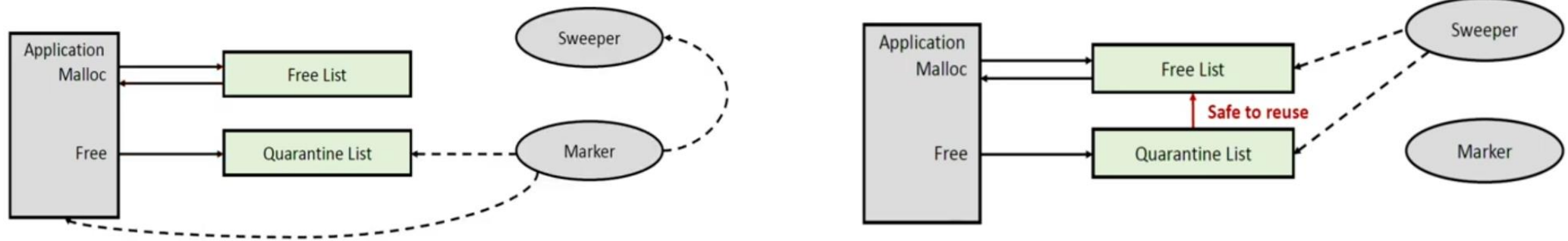


Allocation-intensive Benchmarks

- **Perlbench (SPEC CPU 2006):** 다양한 Perl 스크립트를 실행하여 퍼포먼스 측정
- **Gcc (SPEC CPU 2006):** GNU Compiler Collection의 실제 C 코드 컴파일 테스트
- **Omnetpp (SPEC CPU 2006):** 이산 이벤트 시뮬레이터 (Discrete Event Simulation Framework)
- **Xalancbmk:** SPEC CPU 2006 벤치마크 스위트에 포함된 C++ XML 처리 프로그램 (XSLT transformation)
 - 대표적인 benchmark인 xalancbmk에서 **MarkUs, MineSweeper 모두 2배 이상의 실행 시간 증가**
 - 짧은 시간 내에 많은 청크를 해제하고 할당하는 **Hot spots**이 있음

MarkUs, MineSweeper의 성능 문제

free()된 chunk를 다시 쓰기까지 오래 걸리면,
새 할당은 heap의 더 먼 영역에서 가져오게 됨
→ Hot spots에 치명적



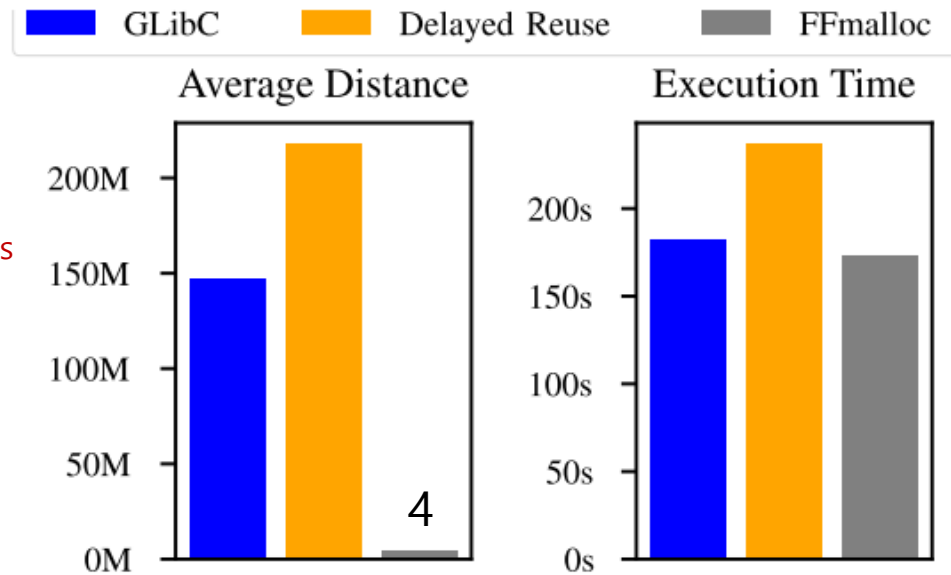
Mark-sweep 방식의 문제점

- Delayed Reuse로 인해 재사용 시점에 따라 chunk들이 메모리 공간에서 멀리 떨어진 주소들로 분산되어 할당됨
 - CPU cache miss 증가, TLB miss 증가, locality 기반 최적화 효과 상실
→ Spatial Locality 저하로 인한 throughput 문제
- MarkUs, MineSweeper는 이러한 방식을 채택했기 때문에 allocation-intensive 애플리케이션에서 성능이 많이 느려짐

- 본 논문에서 allocation-intensive workload에 대한 병목의 원인이 Heap Locality 저하임을 처음으로 지적

I 실험을 통한 검증

The **spatial locality** of the chunks allocated from temporally local allocation requests



The overhead of Delayed Free

■ 실험 방법

- 연속된 50개의 할당에 대해, 각 chunk와 가장 가까운 10개 chunk 사이의 평균 거리를 계산 (=할당 평균 거리)
- 모든 벤치마크 실행 구간에 걸쳐 측정한 후 평균을 냄

- **Never reuse** → **locality** 우수 → **성능 좋음**
- **Delayed reuse**는 공간 지역성을 해치기 때문에, 실제 실행 시간도 느려짐

I 본 논문의 제안: HUSHVAC

▪ Problem

- Mark-sweep 방식에서 **spatial locality 저하**로 인한 throuput 문제

▪ Insight

- FFmalloc처럼 locality를 유지하면 성능 저하가 적다
 - UAF 방지를 위한 Mark-sweep 방식은 유지하되, spatial locality와 성능은 지키자

▪ Solution

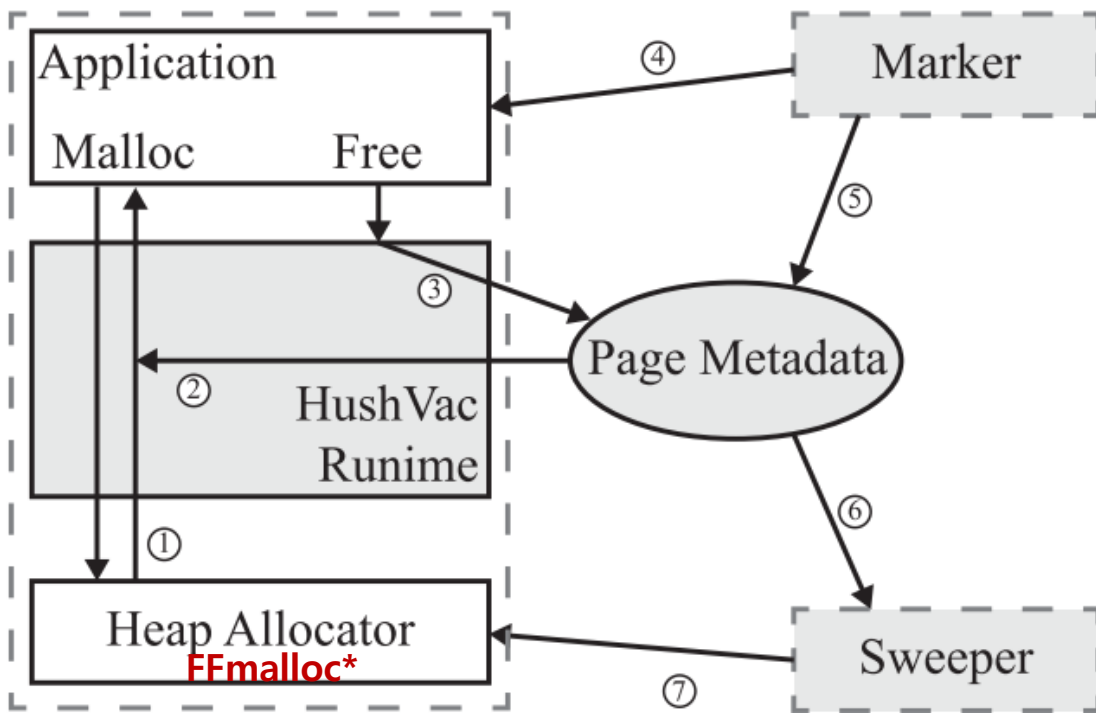
- Opportunistic Page-level Mark-sweep Engine along with FFmalloc
 - **virtual page-level 관리** 및 **opportunistic sweeping** 전략을 적용한 정교한 Heap Allocator 설계



Overview of HUSHVAC

Application Thread (실제 프로그램 실행)

Marker Thread (포인터 추적)



malloc():

- reuse list에 safe page가 있으면 → 그 안에서 할당 ②
- 없으면 → Ffmalloc로 fresh chunk 생성 (mmap 호출) ①

free():

- chunk 상태를 bitmap에 기록 ③
- 페이지 전체가 free 상태면
 - physical frame detach (remap)
 - quarantine list에 추가
- 이후 mark-sweep을 통해 safe하면 reuse list로 이동
 - marker thread가 메모리 전체를 스캔 ④, ⑤
 - sweeper thread가 검사해서 safe 판정 → reuse list ⑥, ⑦

Sweeper Thread
(재사용 가능한 virtual page 결정)

I Overview of HUSHVAC

1

Mark-Sweep For Virtual Pages

→ locality 유지

2

Two-Staged Mark Phase

→ 짧은 stop-the-world

3

Page-Level Sweeping

→ 메모리 usage 절감

4

Opportunistically Triggering the Mark-Sweep Procedure

→ 짧은 stop-the-world

5

Comprehensive Scanning of the Memory Space

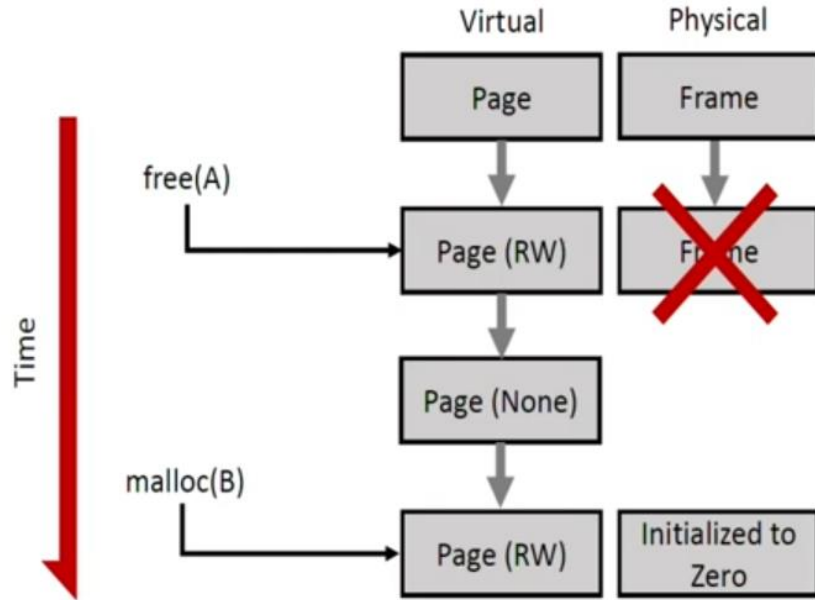
→ anonymous page까지 포괄적으로 스캔

6

Sub-Page Reuse

→ 메모리 usage 최적화

1) Mark-Sweep For Virtual Pages



Release physical memory (remap)

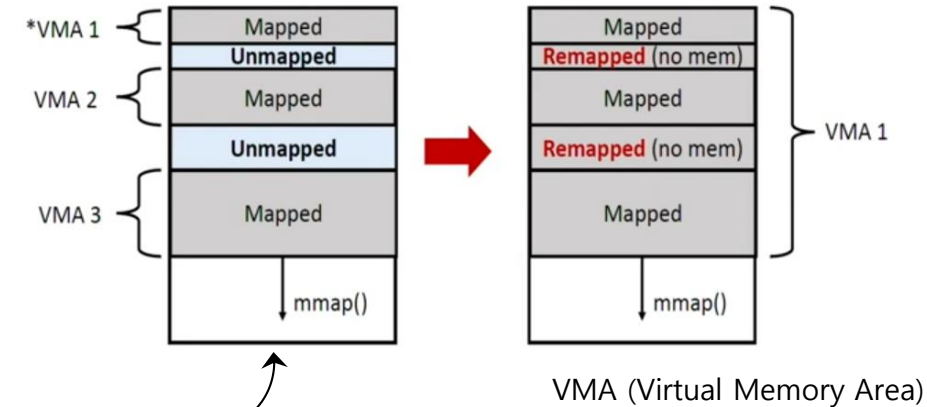
- remap은 **mmap(..., MAP_FIXED)** 방식으로 구현됨
- physical frame은 해제하고 **virtual address는 유지**하기 위해

Delayed to reuse

- 페이지 권한 None → 접근 불가
- dangling pointer를 체크할 때까지 재사용 보류

Safely reused after mark-sweep

- 마킹 결과 안전하다고 판단되면 RW 권한 부여
- 물리 프레임을 새로 **zero-initialized** 상태로 연결



Quarantine List

Reuse Batch List

Chunk 단위가 아니라, Virtual Page 단위로 Mark-sweep 수행

- 하나의 virtual page에 있는 모든 chunk들이 free되었을 때만 mark-sweep 처리
 - 마킹 결과를 page 단위로 집계하여 **sweeping 성능 향상 & locality 유지**
- mmap(..., MAP_FIXED, ...)를 사용하여, 기존 virtual address를 유지한 채 physical frame 연결 해제(detach)
 - 메모리 사용량을 줄이면서도, 실행 시간이나 VMA 수에 큰 오버헤드를 발생시키지 않음**

- freed page는 조만간 다시 사용될 가능성이 높으므로, virtual address를 유지하는 편이 좋음

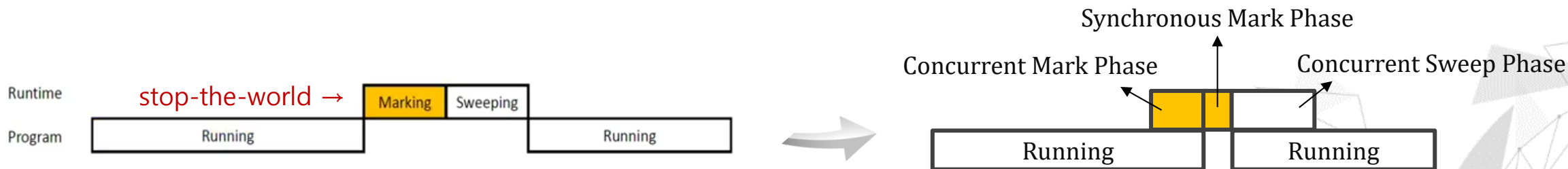
2) Two-Staged Mark Phase

▪ 1단계: Concurrent Mark Phase (비동기 마크)

- 앱은 계속 실행하고 있고, Marker Thread가 전체 메모리를 훑음
- 각 페이지를 순회하면서, 8바이트 단위들 중 heap 내부 주소를 가리키는 값이 있으면, 해당 chunk를 Marking (bitmap)
- 스캔 중 수정된 페이지(dirty page)를 추적
- 대부분의 페이지는 비동기로 처리됨

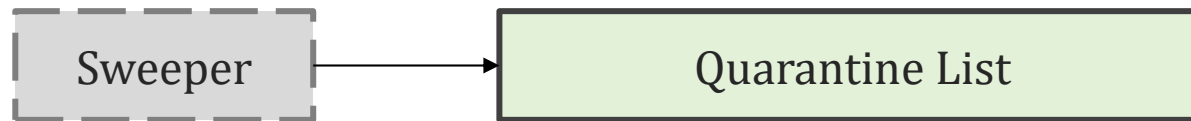
▪ 2단계: Synchronous Mark Phase (동기 마크, stop-the-world)

- 1단계가 끝나면 앱을 잠깐 멈추고, dirty된 페이지만 다시 스캔
- 이 재스캔을 통해 mark map을 정확하게 보정
- 스캔 대상이 줄었기 때문에 stop-the-world 시간이 아주 짧아짐 (MarkUs Avg. 28s / HUSHVAC Avg. 3s)



3) Page-Level Sweeping

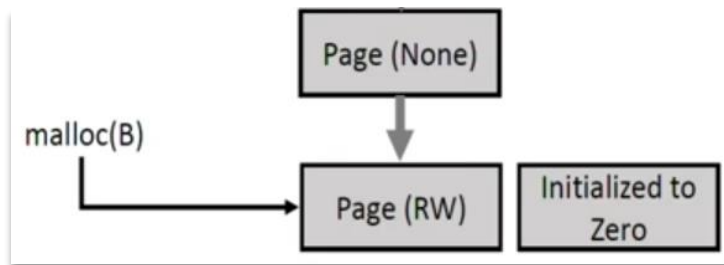
Sweeping thread **runs concurrently** with the application threads



A freed virtual page is considered **safe** to reuse if **all the page's mark bits** remain cleared



Reuse Batch List



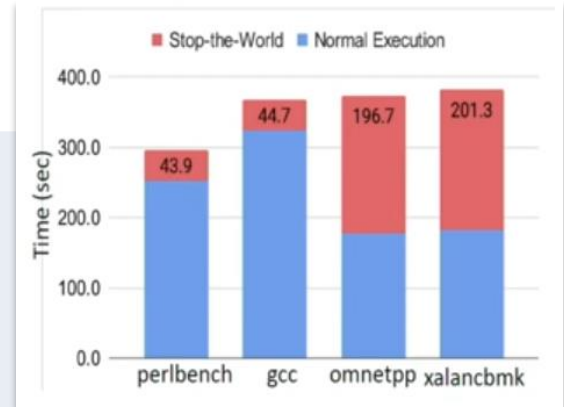
- 다시 읽기/쓰기 권한(PROT_READ | PROT_WRITE) 부여
- mmap으로 새로운 zero-initialized memory 매핑
- 이 페이지는 이제 Ffmalloc의 힙 목록에 재등록됨

- malloc 호출 시 mmap 호출 수가 줄어듦 → **syscall 줄어듦**
- 기존 virtual address 영역을 그대로 사용하므로 → **VMA fragmentation 없음**
- 공간적으로는 이미 물리 메모리를 한 번 release 했기 때문에 → **메모리 오버헤드 감소**

4) Opportunistically Triggering the Mark-Sweep Procedure

기존 mark-sweep 방식의 문제

- 정기적으로 무조건 mark-sweep을 실행
 - stop-the-world time은 그대로 실행 시간 증가로 이어짐
 - 특히 omnetpp, xalancbmk같은 allocation-intensive workload에선 실행 시간의 절반 이상



Heap Allocation 빈도를 관찰하여, 앱이 한가할 때 mark-sweep을 실행하자

- HUSHVAC 런타임 코드를 통해 Application Thread에서 **malloc()**이 호출될 때마다 내부 counter 증가
- Mark-sweep Thread가 주기적으로 동작하면서, 최근 일정 시간 동안 평균 할당 횟수 계산
- 가장 최근의 할당 횟수와 비교하여, 할당 횟수가 **평균의 1.1배보다 적다면** → **mark-sweep 실행**

Quarantine List

- 물리 메모리는 이미 해제됐기 때문에 **quarantine list**가 길어져도 RAM 낭비가 아님
- 따라서, 굳이 지금 mark-sweep하지 않고 앱이 한가할 때 몰래 처리하면 된다.

논문에서 전제하는 프로그램 특성

- 할 일이 많아질수록 메모리를 동적으로 더 많이 할당함
 - 웹 서버 - 응답/요청 데이터 할당
 - 브라우저 - DOM 생성, 레이아웃 객체 동적 할당
- 즉, malloc 수만으로도 앱의 한가한 시점을 유추 가능

5) Comprehensive Scanning of the Memory Space

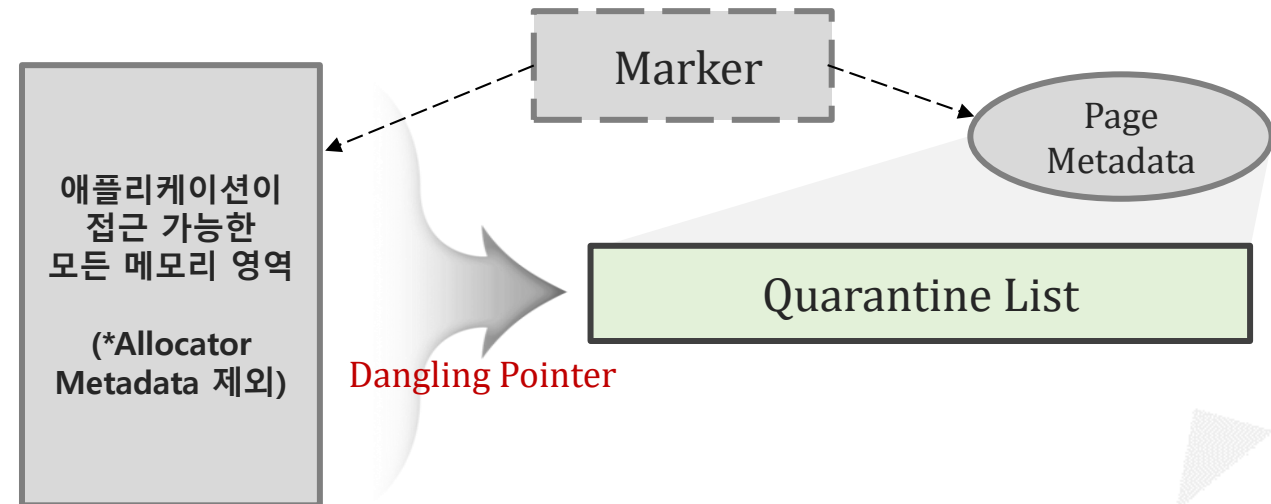
```

1  #define PROT_FLAG    PROT_READ|PROT_WRITE
2  #define MAP_FLAG     MAP_ANON|MAP_PRIVATE
3
4  int main() {
5      void **p = (void **)mmap(NULL, PAGE_SIZE,
6      PROT_FLAG, MAP_FLAG, 0, 0);
7      p[0] = malloc(963751);
8      free(p[0]);
9      p[1] = malloc(963776);
10
11     // [BUG] Reclaim happens: p[0]=0x564549a79000
12     (size=963760) -> p[1]=0x564549a79000 (size
13     =963792)
14     assert(p[0] <= p[1] && p[1] < p[0] + 963760);
15 }

```

HardsHeap이라는 힙 할당기 fuzzer를 수정하여 찾은 PoC

- MarkUs는 mmap된 익명 페이지를 스캔하지 않아서 HardsHeap에 의해 UAF PoC가 생성됨



할당자 메타데이터를 제외한 전체 메모리를 스캔하여 Quarantine List의 가상 페이지를 재사용해도 안전한지 판단

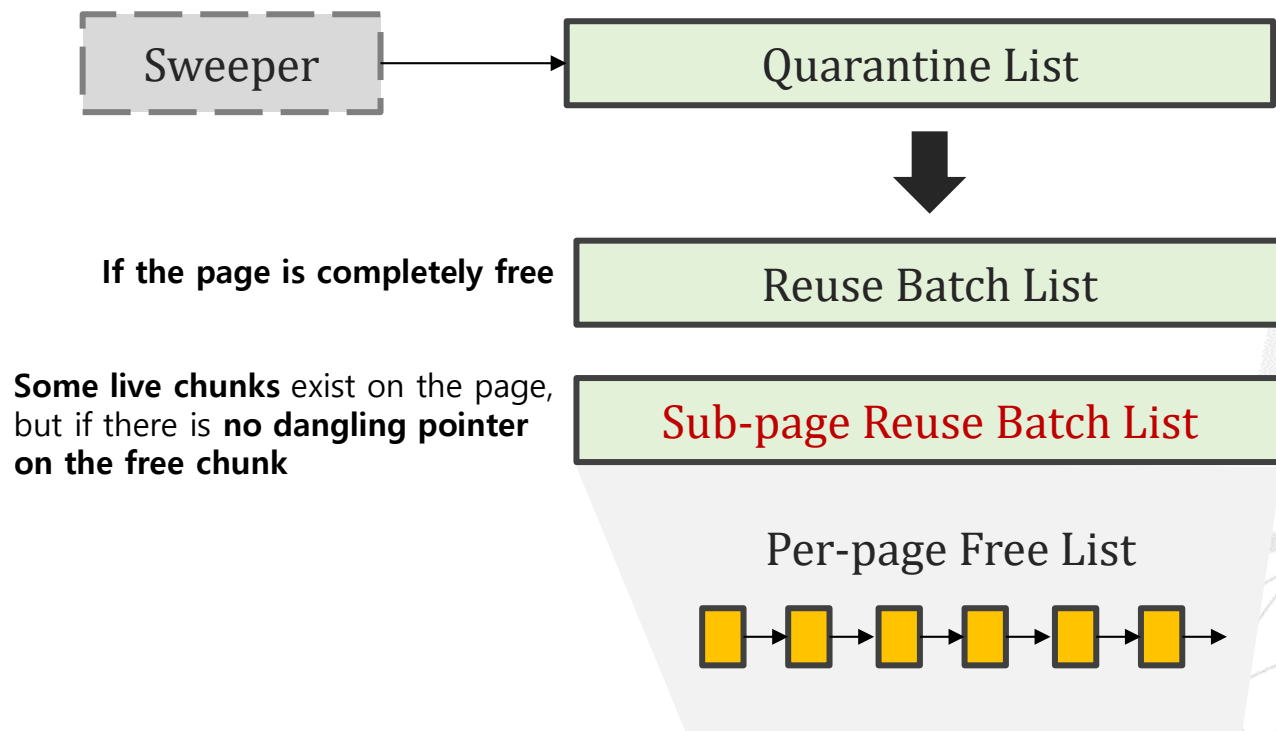
- 애플리케이션이 mmap을 직접 호출하여 검색한 스택, 힙 및 메모리 공간 포함

6) Sub-page Reuse for Mitigating Internal Fragmentation

Internal Fragmentation of Ffmalloc



page-level 관리 원칙은 유지하면서
chunk-level 할당을 잠깐 허용하는 구조



▪ Experimental Setup

- OS/Kernel : Ubuntu 18.04 with Linux 5.4.0-150-generic
- CPU : AMD Ryzen 5 2600
- RAM: 32GB Main Memory
- Thread : HushVac Runs one mark-sweep thread and 10 marker threads

| Performance Benchmarks

▪ SPEC CPU 2006 benchmark suite

- The results we report here will allow the comparison of HUSHVAC with many existing use-after-free prevention schemes.
- -O2 as the default optimization level

▪ SPEC CPU 2017

- the latest version of the SPEC CPU benchmark suite

▪ Firefox의 Bbench

- 현실의 대규모 long-running, interactive GUI 애플리케이션

▪ mimalloc-bench

- Allocator-focused Stress Benchmark

▪ PARSEC 3.0의 C/C++로 작성된 12개의 멀티스레드 워크로드

SPEC CPU 2006

✧ The baseline is glibc

Performance Overhead – (1) Execution time

- **HUSHVAC is 4.7%** on average (geometric mean), which is lower than **MarkUs' 11.4%** but higher than **FFmalloc's -2.1%**
- Regarding **xalancbmk**, HUSHVAC incurs only **35% additional overhead**, whereas MarkUs has a whopping **110% overhead** because of the lesser spatial locality.
 - This result suggests that HUSHVAC is a more attractive allocator that prevents use-after-free when running on the **allocation-intensive benchmarks** than the **existing mark-and-sweep approaches**.

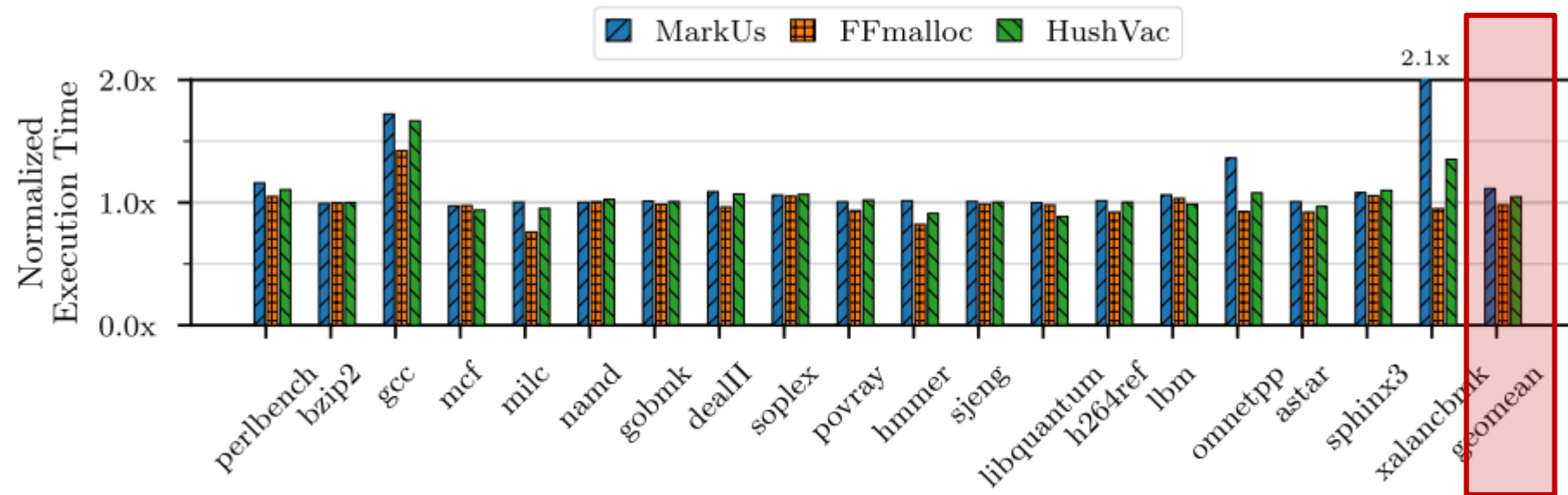


Fig. 4. Normalized execution time when running benchmarks in SPEC CPU 2006 with MarkUs, FFmalloc, and HUSHVAC. MarkUs, FFmalloc, and HUSHVAC slow down the benchmarks by 11.4%, -2.1%, and 4.7% on average (geometric mean), respectively.

| SPEC CPU 2006

▪ Performance Overhead – (2) Stop-the-world delay

- **HUSHVAC** stops the application thread for only **3.03s** on average whereas **MarkUs** pauses for **28s**.
 - Note that FFmalloc does not reuse virtual pages or freed chunks, so it does not stop the application threads.
- When running the well-known **malloc-intensive benchmarks**, such as **perlbench**, **gcc**, **omnetpp**, and **xalancbmk**, **HUSHVAC** exhibits **lower performance overhead** than MarkUs.

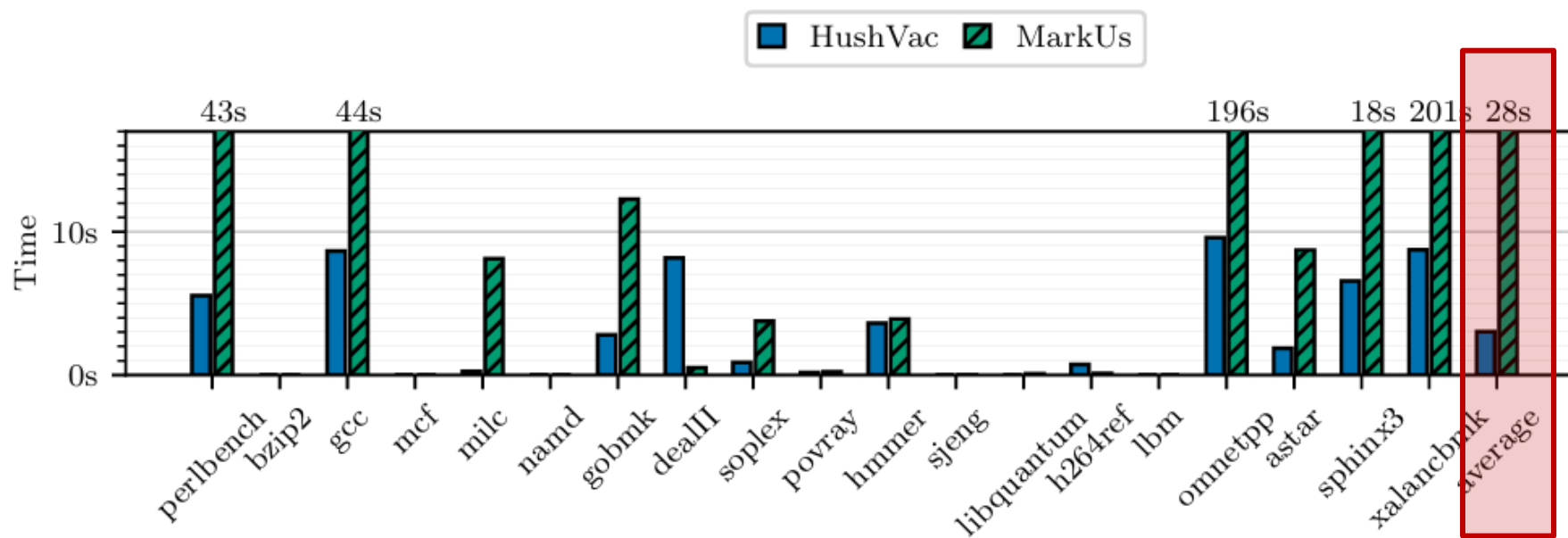


Fig. 5. Stop-the-world delay in SPEC CPU 2006 with MarkUs and HUSHVAC.

SPEC CPU 2006

※ The baseline is glibc

Memory Overhead

- The memory usage **increases** on average (geometric mean) by **25.1% (MarkUs)**, **115% (FFmalloc)**, and **57.2%(HUSHVAC)**.
 - This is less than FFmalloc, whose overhead is 115%, but more than MarkUs, which incurs 25.1% memory overhead only.

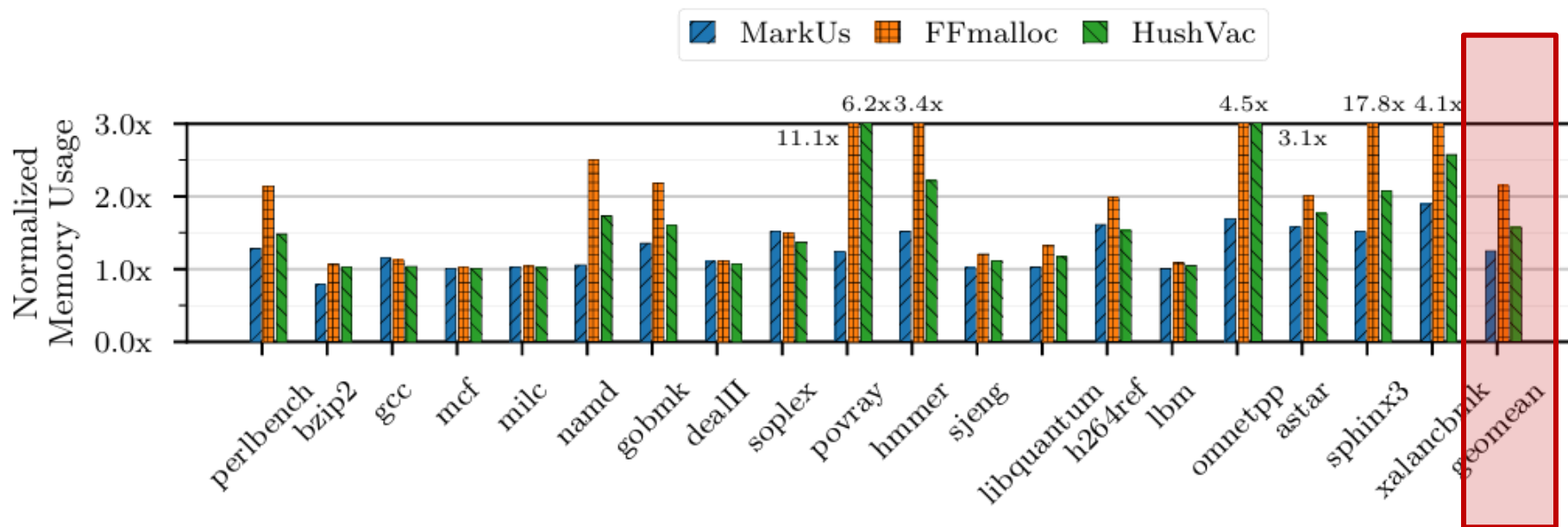
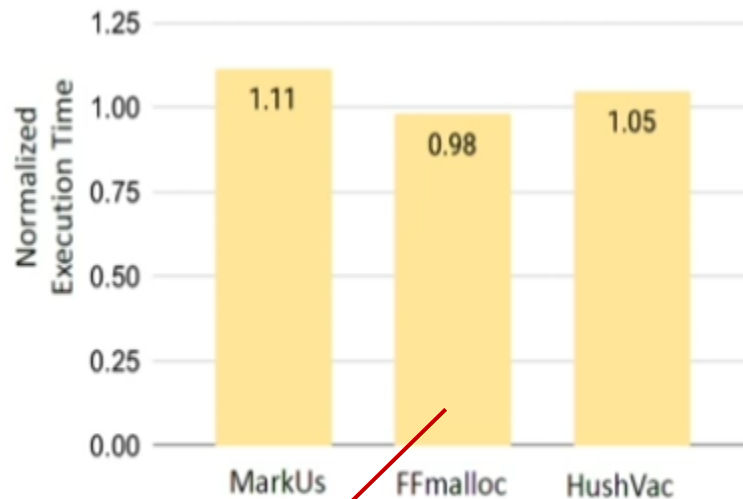


Fig. 6. Normalized memory usage when running benchmarks in SPEC CPU 2006 with MarkUs, FFmalloc, and HUSHVAC. The memory usage increases on average (geometric mean) by 25.1%, 115%, and 57.2% when running with MarkUs, FFmalloc, and HUSHVAC, respectively.

I Average Performance on SPEC 2006

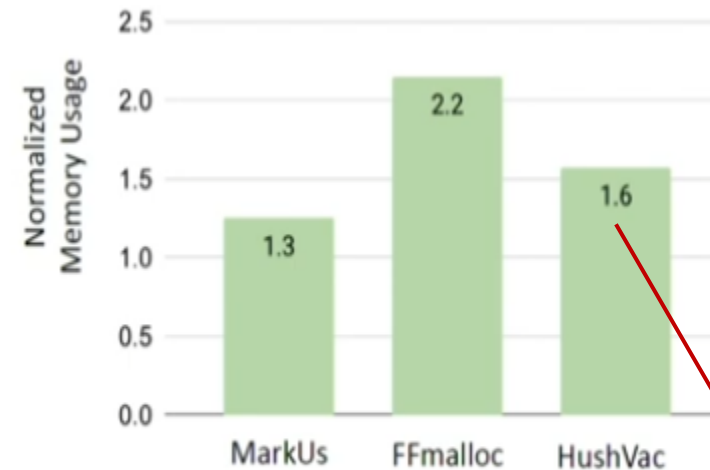
▪ Performance Overhead – (1) Execution time



HUSHVAC은 page 단위로 메모리 관리를 하면서 mmap, munmap 등 system call을 자주 호출함
→ 이로 인해 reuse를 아예 안하는 Ffmalloc보다 7% 느림

HushVac is faster than Markus and has lower memory usage than FFmalloc.

▪ Memory Overhead



internal fragmentation을 완화하기 위해 sub-page reuse를 적용했지만 완벽한 해결책은 아님

Evaluating Design Choices

- most of HUSHVAC's advantage in execution time is owing to the page-level sweeping
 - the two-staged mark phase (TSM) positively affects the execution time on some benchmarks that experience a relatively long stop-the-world delay.

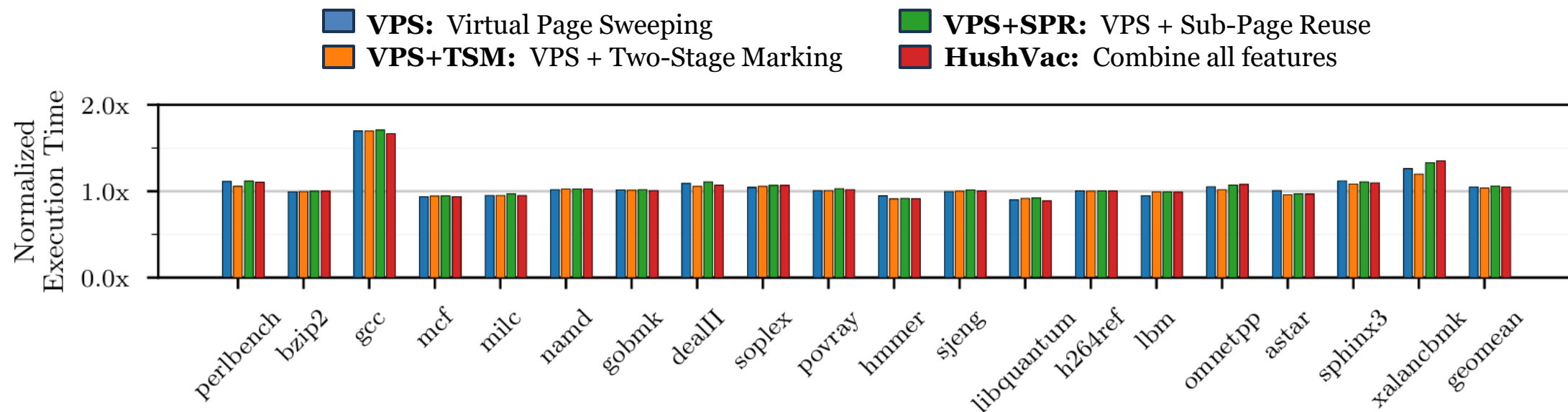


Fig. 19. Normalized execution time when we run benchmarks in SPEC CPU 2006 with variations of HUSHVAC where only a subset of design choices are enabled. VPS stands for virtual page-level sweeping, TSM for two-stage mark phase, and SPR for sub-page reuse.

Evaluating Design Choices

- how the design choices affect the **stop-the world delay**

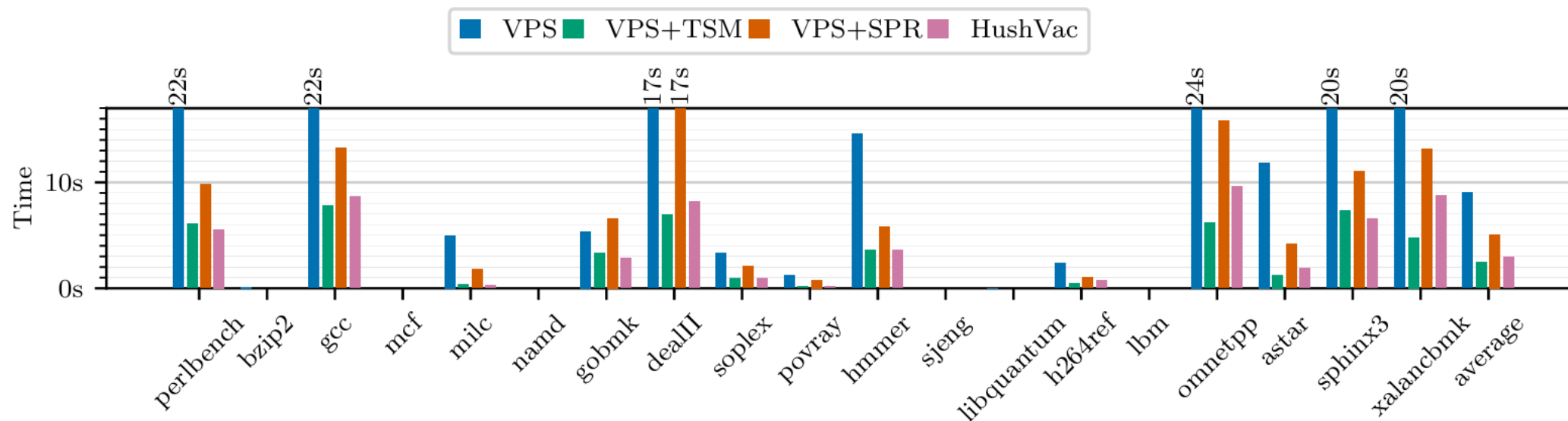


Fig. 20. Stop-the-world delay when we run benchmarks in SPEC CPU 2006 with variations of HUSHVAC where only a subset of design choices are enabled. VPS stands for virtual page-level sweeping, TSM for two-stage mark phase, and SPR for sub-page reuse.

Evaluating Design Choices

- the impact of these design choices on memory usage
 - some benchmarks (e.g., sphinx3) that suffer from a small number of chunks occupying mostly free pages get a significant benefit in memory usage, when we enable sub-page reuse (SPR).

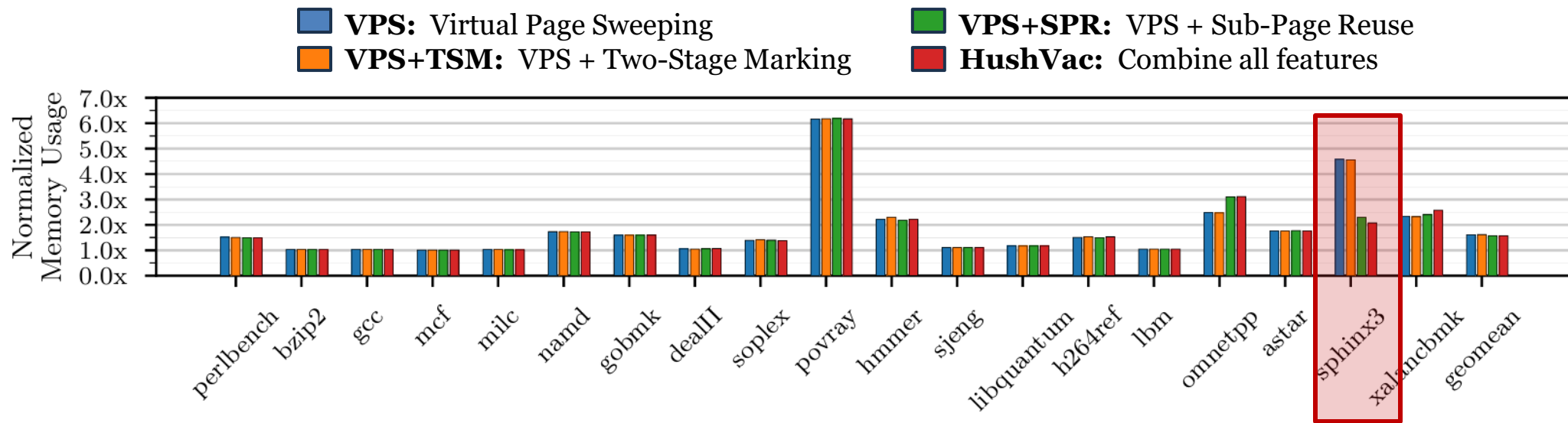
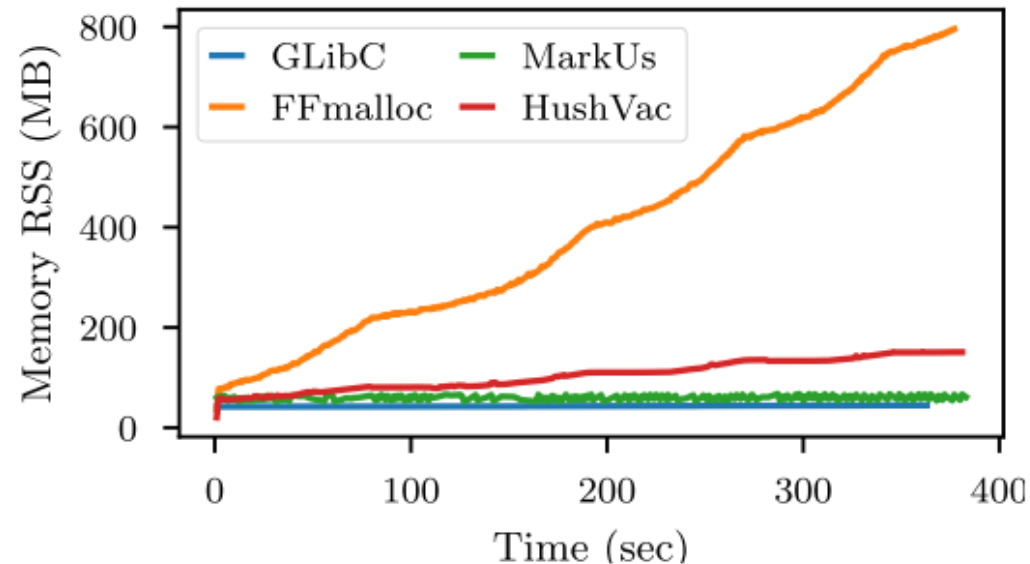


Fig. 21. Normalized Memory usage when we run benchmarks in SPEC CPU 2006 with variations of HUSHVAC where only a subset of design choices are enabled. VPS stands for virtual page-level sweeping, TSM for two-stage mark phase, and SPR for sub-page reuse.

Evaluating Design Choices

Impact of Sub-Page Reuse in Memory Usage

- Unfortunately, the **sub-page reuse (SPR)** feature does not eradicate the trend of increase in memory usage over time, suggesting that sphinx3 has a heap usage pattern, causing a small number of chunks left over for a long time.

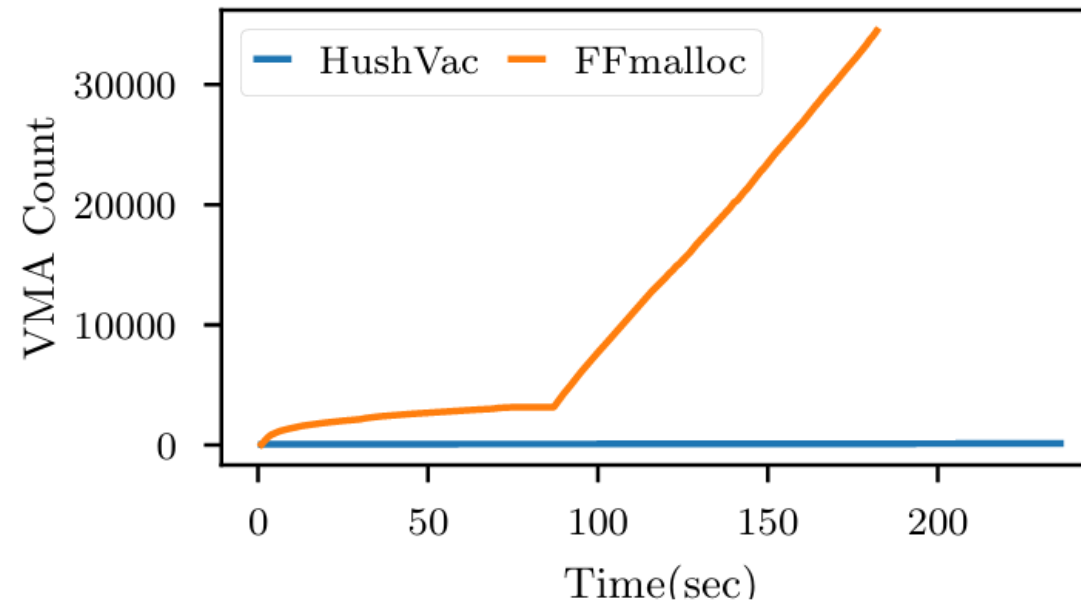


(b) Memory usage over time for sphinx3.

Evaluating Design Choices

Impact on The Number of VMA Structures

- To confirm that our design choice that **detaches physical pages promptly does not cause VMA explosion**, we measure the number of VMA structures over time when running **Xalancbmk**, which is known to **cause a large number of VMA structures**.
- HUSHVAC successfully bounds the number of VMA structures thanks to its design choice of retaining the virtual pages being mapped, while FFmalloc suffers from the VMA explosion problem due to the unmap invocations.



(a) VMA count over time for Xalancbmk.

I Effectiveness in Preventing UAF

- **verify that HUSHVAC can prevent real-world exploits using four public use-after-free exploits**

TABLE I. EFFECTIVENESS OF HUSHVAC IN PREVENTING USE-AFTER-FREE (UAF) EXPLOITS.

Program	CVE ID	Bug Type	Original Attack	With the Protection of HUSHVAC
PHP 7.0.7	CVE-2016-5773 [4]	UAF → double free	Arbitrary code execution	Exploit prevented & double free detected
PHP 5.5.14	CVE-2015-2787 [1]	UAF	Arbitrary code execution	Exploit prevented & runs well
PHP 5.4.44	CVE-2015-6835 [3]	UAF	Arbitrary memory disclosure	Exploit prevented & runs well
PHP 5.4.44	CVE-2015-6834 [2]	UAF	Arbitrary memory disclosure	Exploit prevented & runs well

- **Run HardsHeap's Reclaim module**

- Running HardsHeap for more than 20 hours against HUSHVAC **did not report any working use-after-free examples**, indicating HUSHVAC's effective prevention of use-after-free.

- **HUSHVAC does not abort the use-after-free test cases in the NIST Juliet Test Suite**

- NIST Juliet is a **collection of C/C++ test cases** that includes the ones for testing use-after-free of heap chunks.
- Note that NIST Juliet's use-after-free test cases only evaluates whether **freed chunk are reused using dangling pointers**.
- FFmalloc and HUSHVAC **do not affect this behavior** because they are designed to prevent the use of dangling pointers after a chunk is reused in subsequent allocations, **rather than using freed objects before the reuse**.

▪ Conclusion

- **Simply delaying the reuse** of freed chunk reduces spatial locality.
- garbage collector-like approaches + Giving preference to top chunks, as in the OTA, results in **higher spatial locality**
- HUSHVAC which adopts an innovative approach with **three key design choices** to efficiently avoid use UAF
 - one-time allocator를 기본 할당자로 사용하고, freed virtual pages를 신중하게 재사용하는 접근 방식
 - Page Level Reuse가 실행 가능한 것으로 입증된 이유는 VMA 구조를 분할하지 않고도 **physical page**를 분리하여 개별 **virtual pages**를 격리할 수 있기 때문
 - quarantine list에는 실제 physical page가 없기 때문에 **Opportunistically Triggering the Mark-Sweep Procedure** 가능

감사합니다!