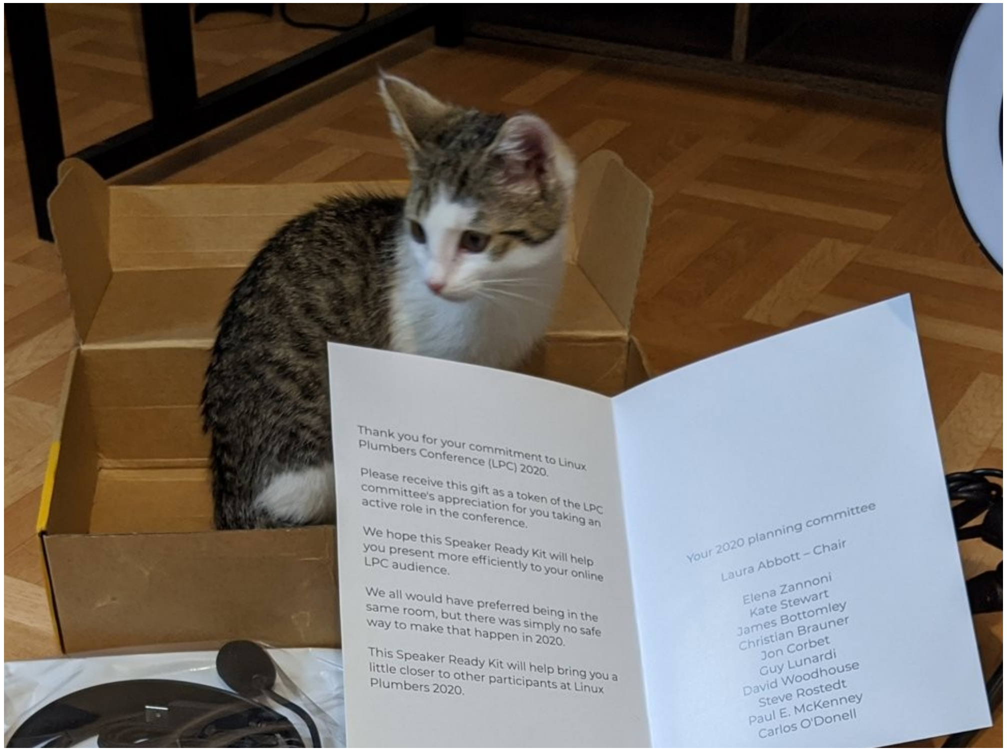
D A M O N:

Data Access MONitoring Framework for Fun and Memory Management Optimizations

SeongJae Park <sjpark@amazon.de>

Disclaimer

- The views expressed herein are those of the speaker;
  they do not reflect the views of his employers
- My cat might come up on the screen. The cat has no ‘`--silent`’ option.
  Sorry, please don’t surprise; keep calm and blame COVID19 :P



https://twitter.com/sjpark0x00/status/1295387149018300419/photo/1

I, SeongJae Park <sjpark@amazon.de>

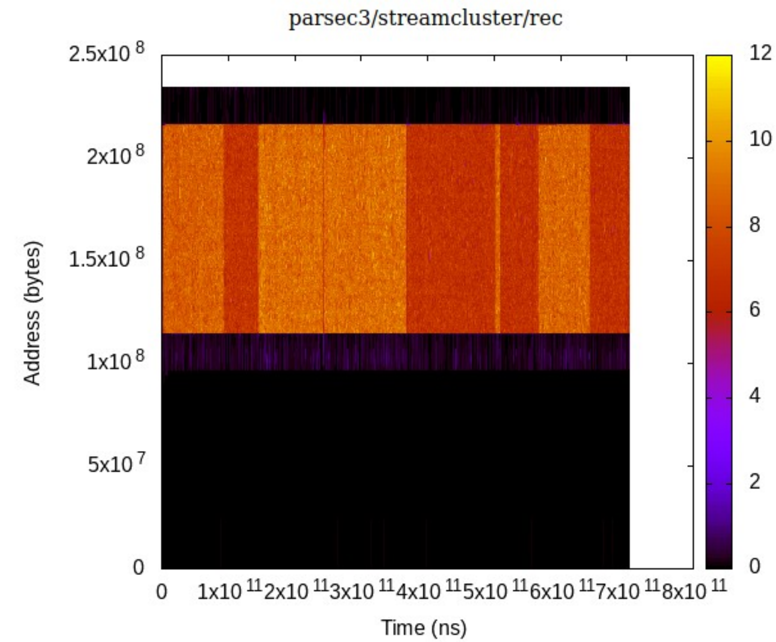
- Kernel / Hypervisor Engineer at Amazon Web Services
- Interested in the memory management and the parallel programming
  - Before joining Amazon, developed
    Guaranteed Contiguous Memory Allocator and
    HTM-based update-side synchronization for RCU on NUMA systems

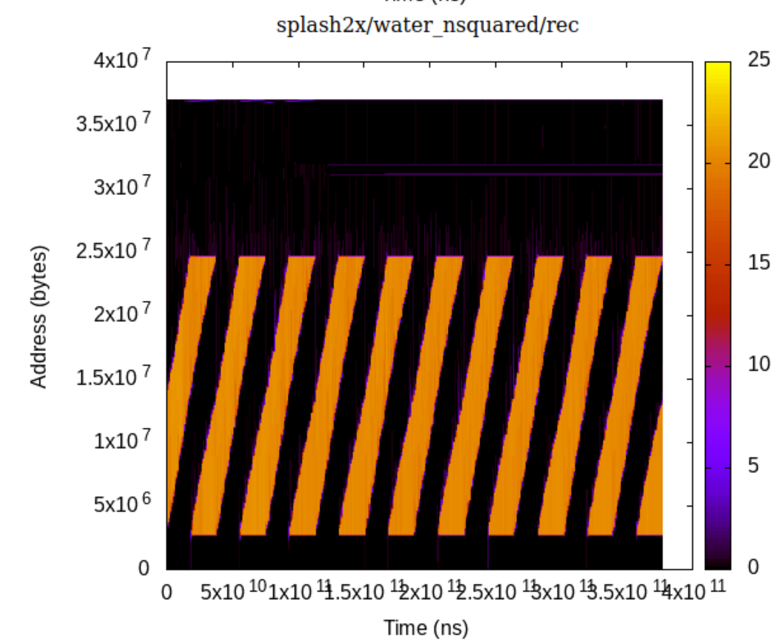
TL; DR

- When: Now
- Who: Sysdmins and kernel subsystems
- Where: On `CONFIG_DAMON=y` kernel
- What: Can monitor the data accesses
- Why: For fun and better MM
- How (for sysadmins):

```
# damo record $(pidof my_workload)
# damo report heats –heatmap a.png;
# eog a.png
```

- Review the patches, please!





Overview

- Motivation
- DAMON
- Live Demo
- Evaluation
- Plans
- Conclusion

Overview

- Motivation
- DAMON
- Live Demo
- Evaluation
- Plans
- Conclusion

Data Access Patterns and The Memory Management

- For a good memory managements, access patterns are needed (e.g., to *keep warm data close and hot data closer* to CPU)
- Linux kernel memory management (MM) gets the information with
  - PTE ‘Accessed’ bits manipulation, fundamentally
- Cons: Coarse
  - Only accessed or not, between some events such as memory pressures
  - 2 LRU lists and heuristics help this a lot
    (The heuristics makes MM more complicated, though)
- Pros: Low overhead
  - Extracting finer information will result in higher overhead
- So, it’s a trade-off that carefully taken. But, wait... is it still the best?

Is MM Happy In This Modern World

- Recent trends might changed some things
  - Working sets are continuously growing
  - DRAM is relatively reducing due to the price and energy consumption
  - New memory devices that slower but cheaper and larger than DRAM evolving
- Based on the trend, a number of optimizations made
  - The works optimize MM in creative ways using finer data access patterns
  - Most of the works show impressive improvements;
Even my simple approach also showed up to 2.55x performance
- But, most of the works had only small interest in the pattern extraction
  - Only naive approaches incurring high/unscalable overhead are used
  - This made most of the works not acceptable in the mainline
  - So, MM needs a subsystem for the fine information, first

Overview

- Motivation
- DAMON
- Live Demo
- Evaluation
- Plans
- Conclusion

Overview

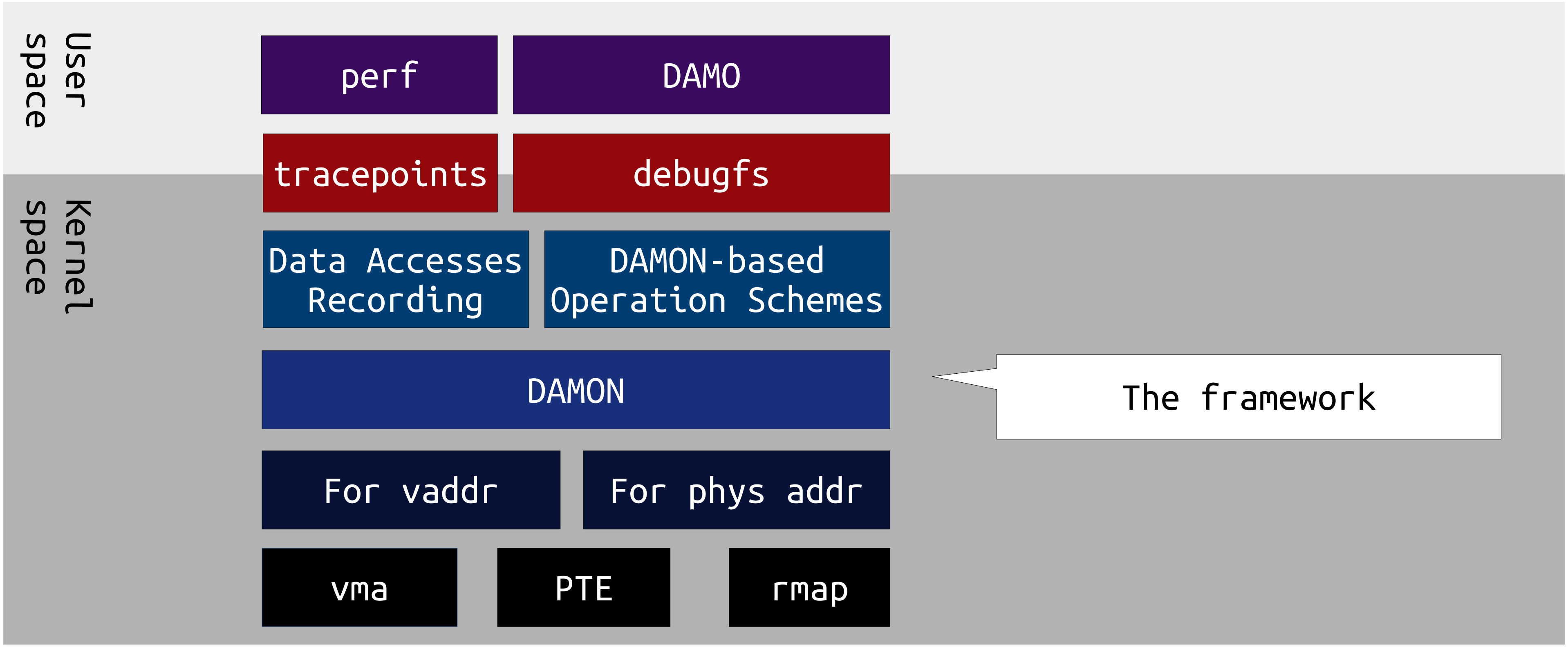
- Motivation
- DAMON
  - What it is and what you can get from it
  - How to use it
  - How it works
  - Misc (how it can be used from user space, how it is tested)
- Live Demo
- Evaluation
- Plans
- Conclusion

Overview

- Motivation
- DAMON
  - What it is and what you can get from it
  - How to use it
  - How it works
  - Misc (how it can be used from user space, how it is tested)
- Live Demo
- Evaluation
- Plans
- Conclusion

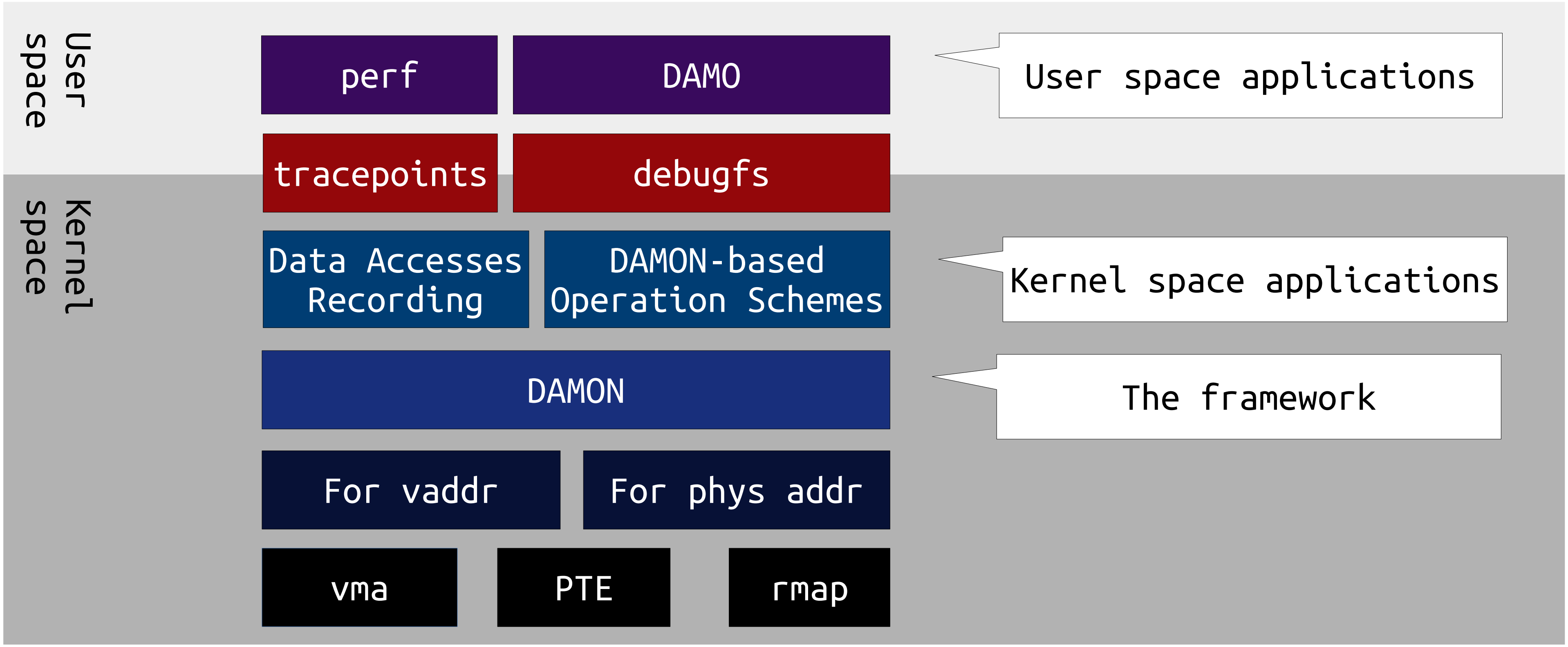
DAMON: Data Access MONitor

- Data access monitoring framework for the Linux kernel
  - Provides access frequency of each address range



DAMON: Data Access MONitor

- Data access monitoring framework for the Linux kernel
  - Provides access frequency of each address range
  - Both kernel space and user space can use it for analysis, memory management optimizations, and fun ;)



Main Design Requirements

- DAMON is desgined to mitigate the overhead-accuracy problem
- For the goal, it fulfills below 4 requirements
  - Accuracy: The monitoring result should be useful for DRAM level MM
  - Overhead: Should light-weight enough for online monitoring
  - Scalability: The upper-bound overhead should be controllable regardless of the size of the monitoring target systems and workloads
  - Generality: The mechanism should be applicable for wide use-cases (e.g., virtual address, physical address, cache-line granularity, ...)

Overview

- Motivation
- DAMON
  - What it is and what you can get from it
  - How to use it
  - How it works
  - Misc (how it can be used from user space, how it is tested)
- Live Demo
- Evaluation
- Plans
- Conclusion

The Usage: Programming Interface

- Step 1: Set the requests in ‘`struct damon_ctx`’ instances
  - How, what memory regions should be monitored
    - The upper-bound monitoring overhead is included here (how)
  - What function should be periodically called back with the results
    - Users can read the monitoring results inside the function
- Step 2: Pass the requests to DAMON via ‘`damon_start()`’
  - Then, a kernel thread for the monitoring is created for each request
  - The thread does monitoring and calls the notification callback
- Unless ‘`damon_start()`’ is called, the system gets no change at all
  - No overhead incurred by just installing `DAMON=y` kernel
  - IOW, No harm `in DAMON=y` ;)

The Usage: In a Pseudo-Code

```
void on_notification(struct damon_ctx *c);

static int __init demo(void)
{
    struct damon_ctx ctx = {
        .aggr_interval = 100,
        .aggregate_cb = on_notification,
        [...]
    };
    damon_start(&ctx, 1);
    msleep(1000*60);
    damon_stop(&ctx, 1);
    return 0;
}

void on_notification(struct damon_ctx *c);
{
    struct damon_region *r;

    damon_for_each_region(r, c)
        pr_info("%lu-%lu accessed %u times during last 100ms\n",
                r->start, r->end, r->nr_accesses);
}
```

Notify me and reset ‘->nr_accesses’ every 100ms

Call ‘on_notification’ for the notification

Overview

- [...]
- DAMON
  - What it is and what you can get from it
  - How to use it
  - How it works
    - control of the overhead and accuracy
    - Support of Various Address Spaces
    - DAMON-based MM optimizations
  - Misc (how it can be used from user space, how it is tested)
- [...]

Overview

- […]
- DAMON
  - What it is and what you can get from it
  - How to use it
  - How it works
    - control of the overhead and accuracy
    - Support of Various Address Spaces
    - DAMON-based MM optimizations
  - Misc (how it can be used from user space, how it is tested)
- […]

Control of Overheads and Monitoring Accuracy

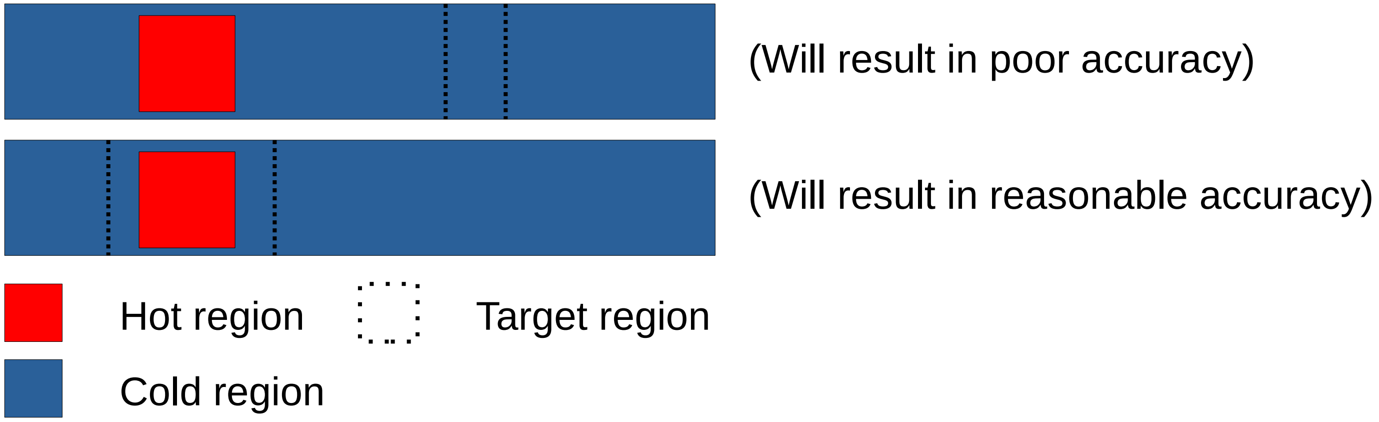
- DAMON becomes DAMON in below steps
  - Straightforward Access Monitoring (Fixed granularity)
  - Region-based Sampling (Elastic granularity)
  - Adaptive Regions Adjustment (Best-effort accuracy)

Straightforward Access Monitoring (Fixed Granularity)

- Periodically check if each monitoring target page is accessed
  - Let’s call the period as ‘sampling interval’
- Aggregate the observations into access frequencies
  - Count number of observed accesses and periodically reset the counter
  - Let’s call the period as ‘aggregation interval’
- By notifying the users just before the reset of the counter, we can provide the access frequency of the pages to the users
- Pros: Fine-grained (page size) monitoring
  - Might not strictly required in some performance-centric optimizations
- Cons: High and unscalable monitoring overhead
  - The overhead arbitrarily increases as the target size grows

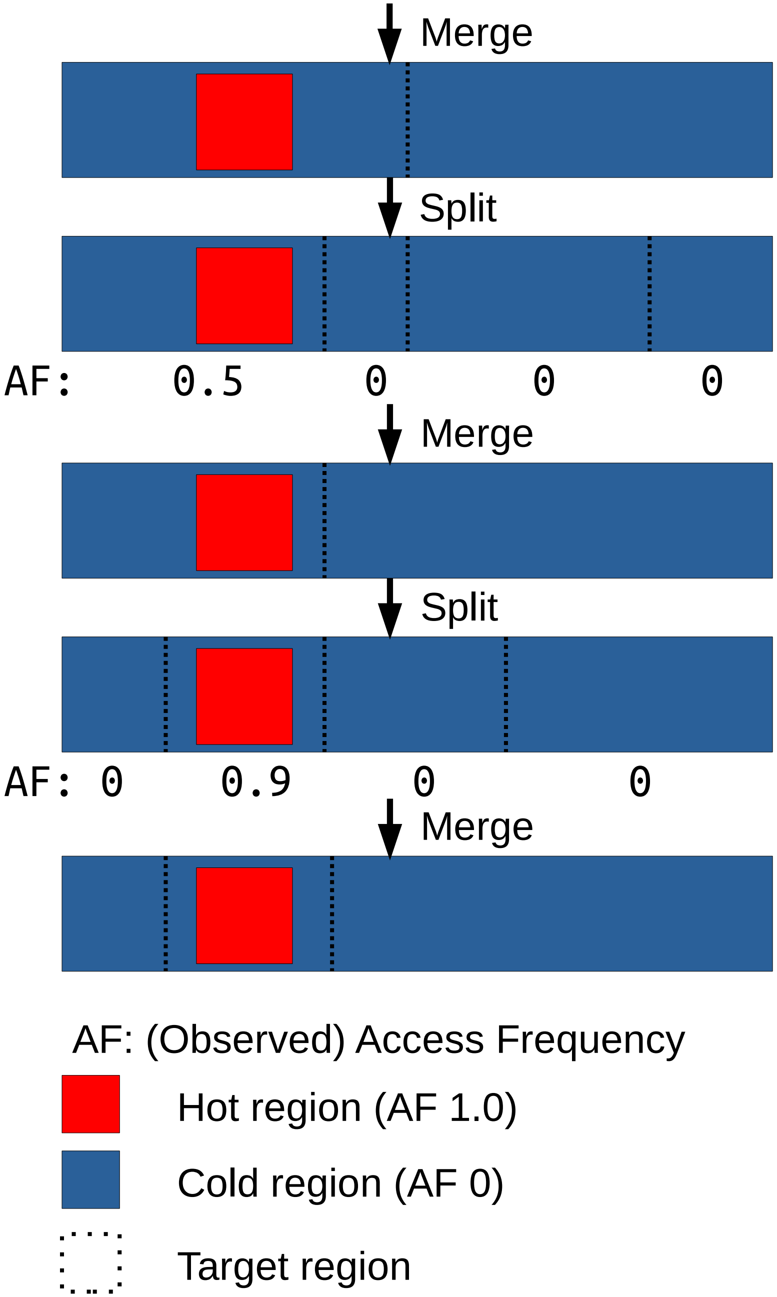
Region-based Sampling (Elastic Granularity)

- Let’s define data objects in access pattern oriented way
  - “A data object is a memory region that all page frames in the region have similar access frequencies”
  - By the definition, if a page in a region is accessed, other pages of the region has probably accessed, and vice versa
  - Thus, checks for the other pages can be skipped
- By limiting the number of regions, we can control the monitoring overhead regardless of the target size
- However, the accuracy will degrade if the regions are not properly set



Adaptive Regions Adjustment (Best-effort Accuracy)

- Starts with minimal number of regions covering entire target memory areas
- For each aggregation interval,
  - merge adjacent regions having similar access frequencies to one region
  - Split each region into two (or three, depend on state) randomly sized regions
  - Avoid merge/split if the number of regions might be out of the user-defined range
- If a split was meaningless, next merge process will revert it (vice versa)
- In this way, we can let users limit the upper bound overhead while preserving best-effort accuracy



Overview

- [...]
- DAMON
  - What it is and what you can get from it
  - How to use it
  - How it works
    - control of the overhead and accuracy
    - Support of Various Address Spaces
    - DAMON-based MM optimizations
  - Misc (how it can be used from user space, how it is tested)
- [...]

Primitives for Fundamental Access Monitoring

- The previous descriptions (intentionally) didn’t explain
  - How the monitoring target regions identified, and
  - How each page is checked whether it has accessed or not
- These two unexplained tasks are
  - Dependent on the detailed use-cases
    - Virtual address space VS physical address space
    - Super high accuracy VS only reasonable accuracy
      (suppose some arch provides dedicated super light access check primitives)
  - Independent with the overhead-accuracy handling core logic

Many Use Cases Are Imaginable

- There are many realistic use cases including
  - Full or parts (e.g., stack) of virtual address spaces of specific processes
  - Full or parts (e.g., NUMA) of physical address spaces of the machine
  - Memory regions backed by specific file or device
  - PTE Accessed bit or LRU position as the access check primitive
  - Dedicated h/w feature of special arch as the access check primitive
- Implementing those in DAMON makes it only complex and inflexible

The Core Logics Are Highly Simple and Flexible

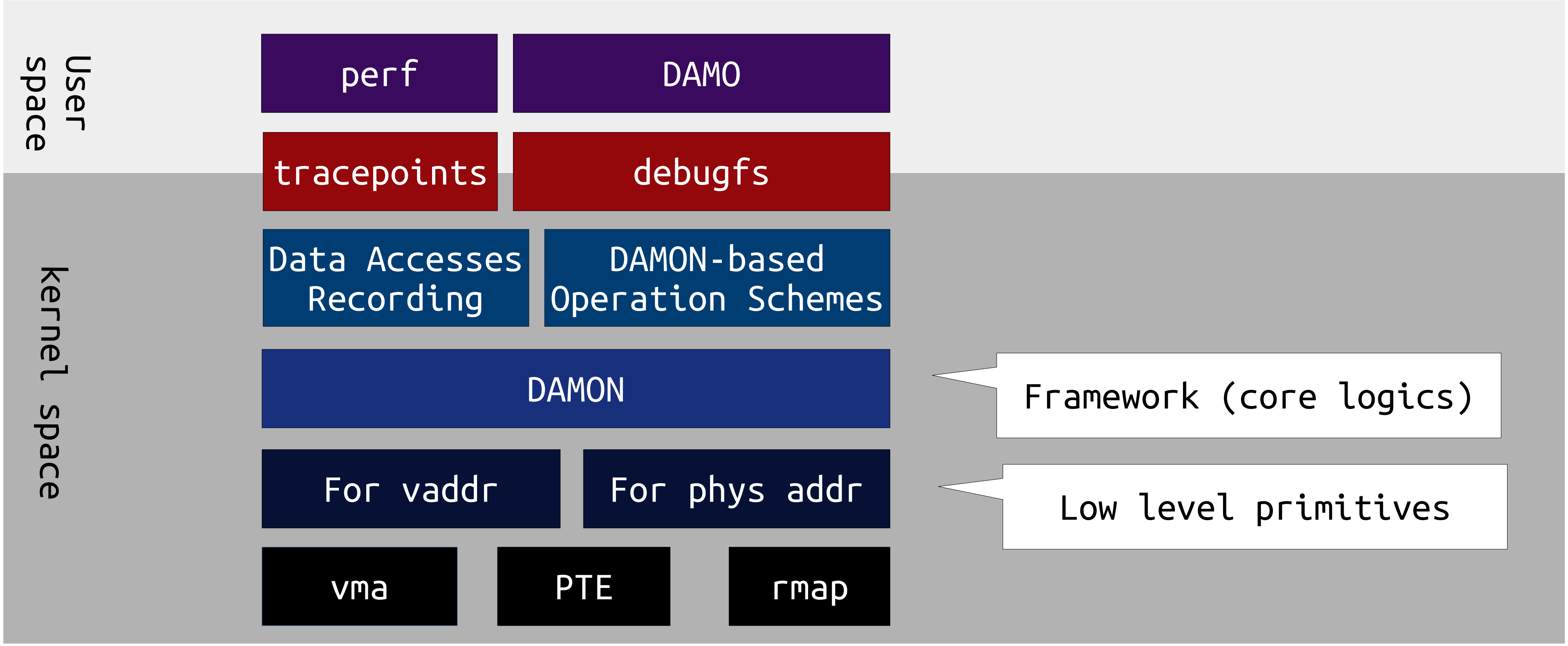
- The minimal access check granularity could be anything
  - The description mentioned page size, but it doesn’t have to be
  - Page size, cache line size, 42 bytes, or even one byte is OK
    if the regions are addressable and access check is possible
- ‘Adaptive regions adjustment' can be turned off
  - Setting the min # of regions same to the max # of regions will turn off the mechanism, as any split and merge will violate the condition
  - Setting the two values same to ``target_size / PAGE_SIZE`` will result in the straightforward page granularity monitoring
- Even multiple CPUs can be used if necessary
  - If you could afford multiple CPUs for super high accuracy, you could
    1) partition the target region into multiple requests and
    2) send those to `damon_start()` at once

Separation of The Primitives and The Core Logic

- The core logic uses the primitives via only cleanly defined interfaces
- The interface is 4 function pointers in the ‘`struct damon_ctx`’
  - `init_regions()`: Initialize the monitoring target regions
  - `update_regions()`: Update the monitoring target regions if there were some changes (e.g., `mmap()` or hot-plug)
  - `prepare_access_check()`:
    Set next sampling target address and mark it not accessed
  - `access_check()`: Check if the sampling target address is accessed
- Any primitives following the interface can be configured to be used

DAMON is Extensible

- DAMON users can extend DAMON for their specific usages
  - Implement own primitives and configure `damon_ctx` to use it
- By default, reference implementations of the primitives for the virtual address spaces and the physical address space are provided



Overview

- [...]
- DAMON
  - What it is and what you can get from it
  - How to use it
  - How it works
    - control of the overhead and accuracy
    - Support of Various Address Spaces
    - DAMON-based MM optimizations
  - Misc (how it can be used from user space, how it is tested)
- [...]

The DAMON-based Optimizations

- Now DAMON-based optimizations are available
  - Both the kernel space and the user space can make the optimization by:
    - Step 1: Run DAMON
    - Step 2: Analyze the monitoring results offline or online
    - Step 3: Make some changes based on the analysis
  - The works unaccepted due to the problem could revisited
- I will optimize the kernel in this way, once DAMON is merged in
  - So the upstream kernels will just work (nearly) optimal someday
  - However, downstream kernel / user space optimizations will still required
    - Because special cases always exist

Risks of Non-upstream DAMON-based Optimizations

- It could be difficult, dangerous, dirty, or restrictive
- Optimizations in the kernel space could be difficult and dangerous
  - Not every sysadmin is experienced kernel programmer
  - Kernel bugs are dangerous
- Optimizations in the user space could be dirty and restrictive
  - Receiving and analyzing the results require some lines of code
    - No library for now; To be honest, I want to do everything on the shell
  - Actions to monitored memory regions are restricted (`madvise_process()` might solve most of this problem)

DAMOS: DAMON-based Operation Schemes

- Yet another kernel feature for easy MM optimizations built on DAMON
- Receives ‘schemes’, each constructed with
  - 3 conditions: Size, access frequency, and age of memory regions
  - 1 memory management action
    - Currently supported actions include: `MADV_(WILLNEED|COLD|PAGEOUT|HUGEPAGE|NOHUGEPAGE)`
- DAMON automatically finds the memory region of the condition and applies the action to the region
- Now users can make DAMON-based optimizations without code

```
# format is:
# <min/max size> <min/max frequency (0-100)> <min/max age> <action>
#
# if a region of size >=4KB didn’t accessed for >=2mins, page out
4K max       0 0       2m max       pageout
```

DAMOS Stats

- Special DAMOS action, ‘`stat`’ does nothing but count
  - Total number of regions matched in the condition
  - Total size of regions matched in the condition
- Users can directly get only meaningful numbers such as
  - Size and number of regions of varying access frequencies and ages
  - No need to get the full results and manually analyze it

```
# format is:
# <min/max size> <min/max frequency (0-100)> <min/max age> <action>
#
min max      0  49        2mins max       stat
min max      50 100       2mins max       stat
min max      0  49        4mins max       stat
min max      50 100       4mins max       stat
```

Overview

- Motivation
- DAMON
  - What it is and what you can get from it
  - How to use it
  - How it works
  - Misc (how it can be used from user space, how it is tested)
- Live Demo
- Evaluation
- Plans
- Conclusion

Interfaces for The User Space

- Tracepoint
  - Provide the monitoring results
  - The monitoring should be manually turned on/off
- DAMON debugfs interface
  - Receive monitoring requests and provide monitoring records
  - Support both the virtual address and physical address
  - The ABI for development of other user space tools
- User space tool: DAMON Operator (DAMO)
  - A reference implementation of user space tool built on the debugfs
  - Provide human friendly interfaces and monitoring results visualization

Tests

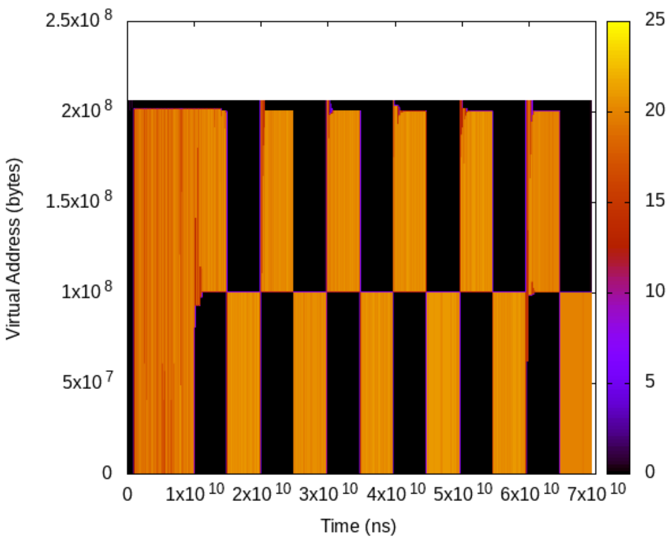
- DAMON has several automated tests for it
- Inside the patchset (should be merged together)
  - User space tests: Test debugfs interface based on kselftest
  - Unit tests: Test internal code based on kunit
- Outside the patchset (would not be merged in the mainline)
  - Correctness tests: Time consuming tests including build/accuracy tests
  - Performance tests: Constructed with 25 realistic workloads
- Outside-the-patchset tests might be open-sourced eventually
  - Hope to be also used as a getting started guide

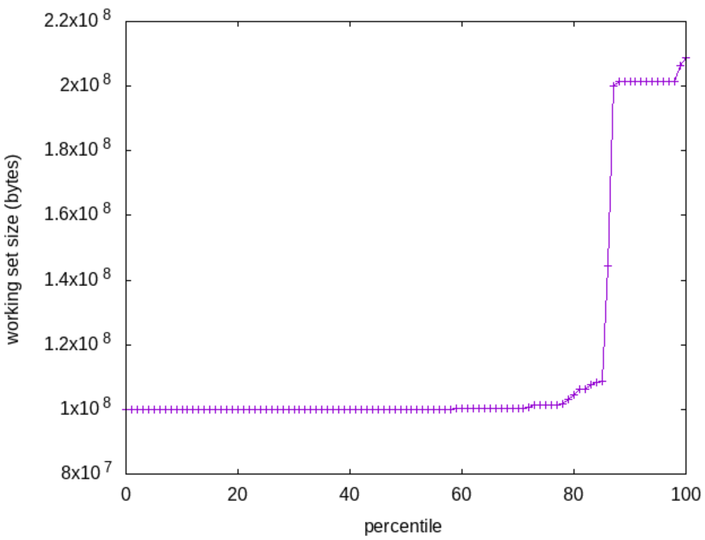
Live Demo using DAMO

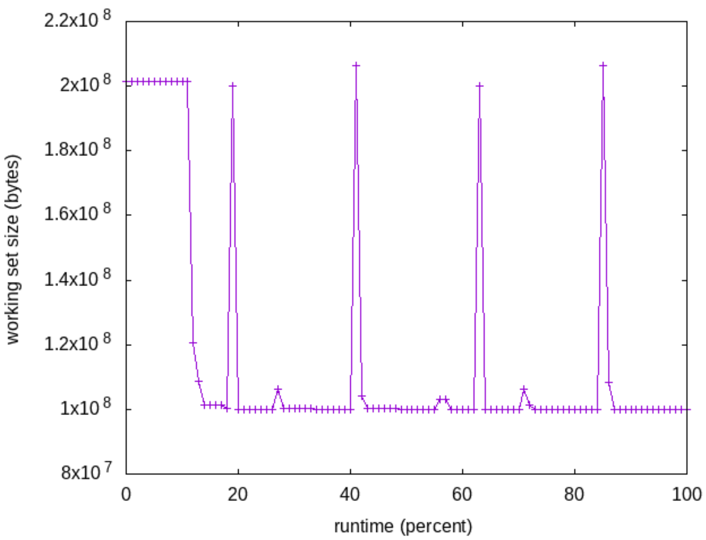
```
$ git clone https://github.com/sjp38/masim
$ cd masim; make; ./masim ./configs/zigzag.cfg &
$ sudo damo record $(pidof masim)

$ damo report raw
$ damo report heats --heatmap access_pattern_heatmap.png
$ damo report wss --range 0 101 1 --plot wss_dist.png
$ damo report wss --range 0 101 1 –sortby time –plot wss_time.png
```

A demo video is also available: https://youtu.be/l63eqbVBZRY







Overview

- Motivation
- DAMON
- Live Demo
- Evaluation
- Plans
- Conclusion

Evaluation Questions

- How lightweight DAMON is?
- How accurate DAMON is?
- How useful DAMON-based optimizations are?

Evaluation Environment

- Test machine
  - QEMU/KVM virtual machine on AWS EC2 i3.metal instance
  - 36 vCPUs, 128 GB memory, 4 GB zram swap device
  - Ubuntu 18.04, THP enabled policy `madvise`
  - Linux v5.8 + DAMON patchsets (The source tree is available)
- Workloads: 25 realistic benchmark workloads
  - 13 workloads from PARSEC3
  - 12 workloads from SPLASH-2X
- DAMON monitoring attributes: The default values
  - 5ms sampling, 100ms aggregation, and 1s regions update intervals
  - Number of regions: [10, 1000]

Evaluation Targets

- Variants
  - orig: DAMON turned off (same to vanilla v5.8)
  - rec: DAMON for virtual address of the workload turned on
  - prec: DAMON for entire physical address of the system turned on
  - thp: THP enabled policy set `always` (to be compared with ethp)
  - ethp: ethp DAMON-based operation scheme is applied

```
$ cat ethp.damos
# for regions having 5/100 access frequency, apply MADV_HUGEPAGE
min max     5 max       min max     hugepage
# for regions >=2MB and not accessed for >=7 seconds, apply MADV_NOHUGEPAGE
2M max      min min     7s max      nohugepage
```

  - prcl: prcl DAMON-based operation scheme is applied

```
$ cat prcl.damos
# for regions >=4KB and not accessed for >=10 seconds, apply MADV_PAGEOUT
4K  max     0 0     10s max     pageout
```

Evaluation Methodology

- Measurement
  - Runtime of the workload
  - System memory usage (`MemTotal – MemFree`)
  - Residential Set Size (RSS) of the workload
  - For each of the workload x variant combinations (25 x 6 = 150)
  - Memory usages are periodically measured and averaged
- Every data is average of 5 different runs (150 x 5 = 750)
  - Run each evaluation on 5 different QEMU VMs on different i3.metal instance
  - Effects from weird outliers might be minimized (still fluctuates, though)
- The test automation code might be open-sourced at last
- Detailed results are available online

Runtime

runtime	orig	rec	(overhead)	prec	(overhead)	thp	(overhead)	ethp	(overhead)	prcl	(overhead)
parsec3/blackscholes	137.688	139.910	(1.61)	138.226	(0.39)	138.524	(0.61)	138.548	(0.62)	150.562	(9.35)
parsec3/bodytrack	124.496	123.294	(-0.97)	124.482	(-0.01)	124.874	(0.30)	123.514	(-0.79)	126.380	(1.51)
parsec3/canneal	196.513	209.465	(6.59)	223.213	(13.59)	189.302	(-3.67)	199.453	(1.50)	242.217	(23.26)
parsec3/dedup	18.060	18.128	(0.38)	18.378	(1.76)	18.210	(0.83)	18.397	(1.87)	20.545	(13.76)
parsec3/facesim	343.697	344.917	(0.36)	341.367	(-0.68)	337.696	(-1.75)	344.805	(0.32)	361.169	(5.08)
parsec3/ferret	288.868	286.110	(-0.95)	292.308	(1.19)	287.814	(-0.36)	284.243	(-1.60)	284.200	(-1.62)
parsec3/fluidanimate	342.267	337.743	(-1.32)	330.680	(-3.39)	337.356	(-1.43)	340.604	(-0.49)	343.565	(0.38)
parsec3/freqmine	437.385	436.854	(-0.12)	437.641	(0.06)	435.008	(-0.54)	436.998	(-0.09)	444.276	(1.58)
parsec3/raytrace	183.036	182.039	(-0.54)	184.859	(1.00)	187.330	(2.35)	185.660	(1.43)	209.707	(14.57)
parsec3/streamcluster	611.075	675.108	(10.48)	656.373	(7.41)	541.711	(-11.35)	473.679	(-22.48)	815.450	(33.45)
parsec3/swaptions	220.338	220.948	(0.28)	220.891	(0.25)	220.387	(0.02)	219.986	(-0.16)	-100.000	(0.00)
parsec3/vips	87.710	88.581	(0.99)	88.423	(0.81)	88.460	(0.86)	88.471	(0.87)	89.661	(2.22)
parsec3/x264	114.927	117.774	(2.48)	116.630	(1.48)	112.237	(-2.34)	110.709	(-3.67)	124.560	(8.38)
splash2x/barnes	131.034	130.895	(-0.11)	129.088	(-1.48)	118.213	(-9.78)	124.497	(-4.99)	167.966	(28.19)
splash2x/fft	59.805	60.237	(0.72)	59.895	(0.15)	47.008	(-21.40)	57.962	(-3.08)	87.183	(45.78)
splash2x/lu_cb	132.353	132.157	(-0.15)	132.473	(0.09)	131.561	(-0.60)	135.541	(2.41)	141.720	(7.08)
splash2x/lu_ncb	149.050	150.496	(0.97)	151.912	(1.92)	150.974	(1.29)	148.329	(-0.48)	152.227	(2.13)
splash2x/ocean_cp	82.189	77.735	(-5.42)	84.466	(2.77)	77.498	(-5.71)	82.586	(0.48)	113.737	(38.38)
splash2x/ocean_ncp	154.934	154.656	(-0.18)	164.204	(5.98)	101.861	(-34.26)	142.600	(-7.96)	281.650	(81.79)
splash2x/radiosity	142.710	141.643	(-0.75)	143.940	(0.86)	141.982	(-0.51)	142.017	(-0.49)	152.116	(6.59)
splash2x/radix	50.357	50.331	(-0.05)	50.717	(0.72)	45.664	(-9.32)	50.222	(-0.27)	73.981	(46.91)
splash2x/raytrace	134.039	132.650	(-1.04)	134.583	(0.41)	131.570	(-1.84)	133.050	(-0.74)	141.463	(5.54)
splash2x/volrend	120.769	120.220	(-0.45)	119.895	(-0.72)	120.159	(-0.50)	119.311	(-1.21)	119.581	(-0.98)
splash2x/water_nsquared	376.599	373.411	(-0.85)	382.601	(1.59)	348.701	(-7.41)	357.033	(-5.20)	397.427	(5.53)
splash2x/water_spatial	132.619	133.432	(0.61)	135.505	(2.18)	134.865	(1.69)	133.940	(1.00)	148.196	(11.75)
total	4772.510	4838.740	(1.39)	4862.740	(1.89)	4568.970	(-4.26)	4592.160	(-3.78)	5189.560	(8.74)

This is also available online.
Summarized analysis is in following slide,
so you don’t need to read this now.

System Memory Usage (MemTotal - MemFree)

memused.avg	orig	rec	(overhead)	prec	(overhead)	thp	(overhead)	ethp	(overhead)	prcl	(overhead)
parsec3/blackscholes	1825022.800	1863815.200	(2.13)	1830082.000	(0.28)	1800999.800	(-1.32)	1807743.800	(-0.95)	1580027.800	(-13.42)
parsec3/bodytrack	1425506.800	1438323.400	(0.90)	1439260.600	(0.96)	1400505.600	(-1.75)	1412295.200	(-0.93)	1412759.600	(-0.89)
parsec3/canneal	1040902.600	1050404.000	(0.91)	1053535.200	(1.21)	1027175.800	(-1.32)	1035229.400	(-0.55)	1039159.400	(-0.17)
parsec3/dedup	2526700.400	2540671.600	(0.55)	2503689.800	(-0.91)	2544440.200	(0.70)	2510519.000	(-0.64)	2503148.200	(-0.93)
parsec3/facesim	545844.600	550680.000	(0.89)	543658.600	(-0.40)	532320.200	(-2.48)	539429.600	(-1.18)	470836.800	(-13.74)
parsec3/ferret	352118.600	326782.600	(-7.20)	322645.600	(-8.37)	304054.800	(-13.65)	317259.000	(-9.90)	313532.400	(-10.96)
parsec3/fluidanimate	651597.600	580045.200	(-10.98)	578297.400	(-11.25)	569431.600	(-12.61)	577322.800	(-11.40)	482061.600	(-26.02)
parsec3/freqmine	989212.000	996291.200	(0.72)	989405.000	(0.02)	970891.000	(-1.85)	981122.000	(-0.82)	736030.000	(-25.59)
parsec3/raytrace	1749470.400	1751183.200	(0.10)	1740937.600	(-0.49)	1717138.800	(-1.85)	1731298.200	(-1.04)	1528069.000	(-12.66)
parsec3/streamcluster	123425.400	151548.200	(22.79)	144024.800	(16.69)	118379.000	(-4.09)	124845.400	(1.15)	118629.800	(-3.89)
parsec3/swaptions	4150.600	25679.200	(518.69)	19914.800	(379.80)	8577.000	(106.64)	17348.200	(317.97)	-100.000	(0.00)
parsec3/vips	2989801.200	3003285.400	(0.45)	3012055.400	(0.74)	2958369.000	(-1.05)	2970897.800	(-0.63)	2962063.000	(-0.93)
parsec3/x264	3242663.400	3256091.000	(0.41)	3248949.400	(0.19)	3195605.400	(-1.45)	3206571.600	(-1.11)	3219046.333	(-0.73)
splash2x/barnes	1208017.600	1212702.600	(0.39)	1194143.600	(-1.15)	1208450.200	(0.04)	1212607.600	(0.38)	878554.667	(-27.27)
splash2x/fft	9786259.000	9705563.600	(-0.82)	9391006.800	(-4.04)	9967230.600	(1.85)	9657639.400	(-1.31)	10215759.333	(4.39)
splash2x/lu_cb	512130.400	521431.800	(1.82)	513051.400	(0.18)	508534.200	(-0.70)	512643.600	(0.10)	328017.333	(-35.95)
splash2x/lu_ncb	511156.200	526566.400	(3.01)	513230.400	(0.41)	509823.800	(-0.26)	516302.000	(1.01)	418078.333	(-18.21)
splash2x/ocean_cp	3353269.200	3319496.000	(-1.01)	3251575.000	(-3.03)	3379639.800	(0.79)	3326416.600	(-0.80)	3143859.667	(-6.24)
splash2x/ocean_ncp	3905538.200	3914929.600	(0.24)	3877493.200	(-0.72)	7053949.400	(80.61)	4633035.000	(18.63)	3527482.667	(-9.68)
splash2x/radiosity	1462030.400	1468050.000	(0.41)	1454997.600	(-0.48)	1466985.400	(0.34)	1461777.400	(-0.02)	441332.000	(-69.81)
splash2x/radix	2367200.800	2363995.000	(-0.14)	2251124.600	(-4.90)	2417603.800	(2.13)	2317804.000	(-2.09)	2495581.667	(5.42)
splash2x/raytrace	42356.200	56270.200	(32.85)	49419.000	(16.67)	86408.400	(104.00)	50547.600	(19.34)	40341.000	(-4.76)
splash2x/volrend	148631.600	162954.600	(9.64)	153305.200	(3.14)	140089.200	(-5.75)	149831.200	(0.81)	150232.000	(1.08)
splash2x/water_nsquared	39835.800	54268.000	(36.23)	53659.400	(34.70)	41073.600	(3.11)	85322.600	(114.19)	49463.667	(24.17)
splash2x/water_spatial	669746.600	679634.200	(1.48)	667518.600	(-0.33)	664383.800	(-0.80)	684470.200	(2.20)	401946.000	(-39.99)
total	41472600.000	41520700.000	(0.12)	40796900.000	(-1.63)	44592000.000	(7.52)	41840100.000	(0.89)	38456146.000	(-7.27)

This is also available online.
Summarized analysis is in following slide,
so you don’t need to read this now.

Residential Set Size (RSS)

rss.avg	orig	rec	(overhead)	prec	(overhead)	thp	(overhead)	ethp	(overhead)	prcl	(overhead)
parsec3/blackscholes	587078.800	586930.400	(-0.03)	586355.200	(-0.12)	586147.400	(-0.16)	585203.400	(-0.32)	243110.800	(-58.59)
parsec3/bodytrack	32470.800	32488.400	(0.05)	32351.000	(-0.37)	32433.400	(-0.12)	32429.000	(-0.13)	18804.800	(-42.09)
parsec3/canneal	842418.600	842442.800	(0.00)	844396.000	(0.23)	840756.400	(-0.20)	841242.000	(-0.14)	825296.200	(-2.03)
parsec3/dedup	1180100.000	1179309.200	(-0.07)	1160477.800	(-1.66)	1198789.200	(1.58)	1171802.600	(-0.70)	595531.600	(-49.54)
parsec3/facesim	312056.000	312109.200	(0.02)	312044.400	(-0.00)	318102.200	(1.94)	316239.600	(1.34)	192002.600	(-38.47)
parsec3/ferret	99792.200	99641.800	(-0.15)	99044.800	(-0.75)	102041.800	(2.25)	100854.000	(1.06)	83628.200	(-16.20)
parsec3/fluidanimate	530735.400	530759.000	(0.00)	530865.200	(0.02)	532440.800	(0.32)	522778.600	(-1.50)	433547.400	(-18.31)
parsec3/freqmine	552951.000	552788.000	(-0.03)	552761.800	(-0.03)	556004.400	(0.55)	554001.200	(0.19)	47881.200	(-91.34)
parsec3/raytrace	883966.600	880061.400	(-0.44)	883144.800	(-0.09)	871786.400	(-1.38)	881000.200	(-0.34)	267210.800	(-69.77)
parsec3/streamcluster	110901.600	110863.400	(-0.03)	110893.600	(-0.01)	115612.600	(4.25)	114976.800	(3.67)	109728.600	(-1.06)
parsec3/swaptions	5708.800	5712.400	(0.06)	5681.400	(-0.48)	5720.400	(0.20)	5726.000	(0.30)	-100.000	(0.00)
parsec3/vips	32272.200	32427.400	(0.48)	31959.800	(-0.97)	34177.800	(5.90)	33306.400	(3.20)	28869.000	(-10.55)
parsec3/x264	81878.000	81914.200	(0.04)	81823.600	(-0.07)	83579.400	(2.08)	83236.800	(1.66)	81220.667	(-0.80)
splash2x/barnes	1211917.400	1211328.200	(-0.05)	1212450.400	(0.04)	1221951.000	(0.83)	1218924.600	(0.58)	489430.333	(-59.62)
splash2x/fft	9874359.000	9934912.400	(0.61)	9843789.600	(-0.31)	10204484.600	(3.34)	9980640.400	(1.08)	7003881.000	(-29.07)
splash2x/lu_cb	509066.200	509222.600	(0.03)	509059.600	(-0.00)	509594.600	(0.10)	509479.000	(0.08)	315538.667	(-38.02)
splash2x/lu_ncb	509192.200	508437.000	(-0.15)	509331.000	(0.03)	509606.000	(0.08)	509578.200	(0.08)	412065.667	(-19.07)
splash2x/ocean_cp	3380283.800	3380301.000	(0.00)	3377617.200	(-0.08)	3416531.200	(1.07)	3389845.200	(0.28)	2398084.000	(-29.06)
splash2x/ocean_ncp	3917913.600	3924529.200	(0.17)	3934911.800	(0.43)	7123907.400	(81.83)	4703623.600	(20.05)	2428288.000	(-38.02)
splash2x/radiosity	1467978.600	1468655.400	(0.05)	1467534.000	(-0.03)	1477722.600	(0.66)	1471036.000	(0.21)	148573.333	(-89.88)
splash2x/radix	2413933.400	2408367.600	(-0.23)	2381122.400	(-1.36)	2480169.400	(2.74)	2367118.800	(-1.94)	1848857.000	(-23.41)
splash2x/raytrace	23280.000	23272.800	(-0.03)	23259.000	(-0.09)	28715.600	(23.35)	28354.400	(21.80)	13302.333	(-42.86)
splash2x/volrend	44079.400	44091.600	(0.03)	44022.200	(-0.13)	44547.200	(1.06)	44615.600	(1.22)	29833.000	(-32.32)
splash2x/water_nsquared	29392.800	29425.600	(0.11)	29422.400	(0.10)	30317.800	(3.15)	30602.200	(4.11)	21769.000	(-25.94)
splash2x/water_spatial	658604.400	660276.800	(0.25)	660334.000	(0.26)	660491.000	(0.29)	660636.400	(0.31)	304246.667	(-53.80)
total	29292400.000	29350400.000	(0.20)	29224634.000	(-0.23)	32985491.000	(12.61)	30157300.000	(2.95)	18340700.000	(-37.39)

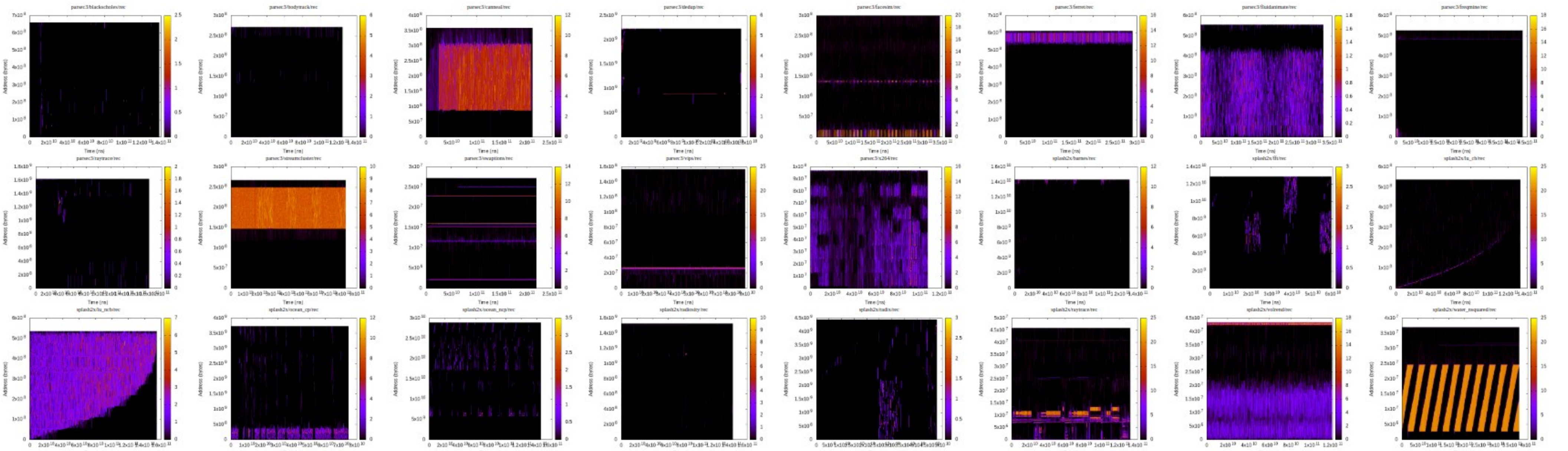
This is also available online.
Summarized analysis is in following slide,
so you don’t need to read this now.

Overhead is Modest

- Normally only 0.3 `%CPU` is used by the monitoring thread
- In total, virtual/physical address monitoring respectively incur
  - 1.39% / 1.89% target workload slowdown
  - 0.12% / -1.63% memory usage overhead
  - 0.20% / -0.23% RSS overhead
  - Note the small diff between two, despite of the big target size difference
    - 128GB for physical, 6MB-9GB for virtual address space
    - The target size doesn’t affect the overhead, as promised

Monitoring Results Seems Reasonably Accurate

- Various visualizations of the monitoring results including
  - Heatmap
  - Working set size distribution based on size and time
  - Number of (adaptively constructed) monitoring target regions
- All cleanly show reasonable access patterns and distributions



This is also available online.

DAMOS: Access-aware THP Hints (`ethp`)

- ‘`ethp`’ preserves the speedup while reducing the memory bloat
- Interesting case: `splash2x/ocean_ncp`
  - `thp` achieves 34.26% speedup but 80.61% system memory waste (best speedup and worst memory waste among the 25 workloads)
  - `ethp` achieves 7.96% speedup but only 18.63% system memory waste
  - Hence, `ethp` removes 76.90% of memory waster while preserving 23.23% of speedup for the workload
- Removes 88.16% of `thp`’s system memory waste in total
- Preserves 88.73% of `thp`’s speedup in total
- NOTE: `ethp` is only for proof-of-concept and thus not optimized
  - The ethp speedup could be higher if khugepaged promote pages marked with MADV_HUGEPAGE more aggressively

DAMOS: Proactive Reclamation (`prcl`)

- ‘`prcl`’ reduces working sets while making only modest slowdown
- Incurs 8.74% speed down in total
- Reduces 37.39% RSS in total
- Best case: `parsec3/freqmine`
  - Recuces 91.34% of RSS while incurring only 1.58% speed down
- NOTE: `prcl` is only for proof of concept and thus not optimized
  - Paging out 10sec inactive pages might be too aggressive (Google’s proactive reclamation waits 2 minutes)
  - With faster swap devices, the speed down could further reduced

Evaluation Wrap-up: DAMON Is...

- Lightweight
- Accurate
- Useful for MM optimizations

Overview

- Motivation
- DAMON
- Live Demo
- Evaluation
- Plans
- Conclusion

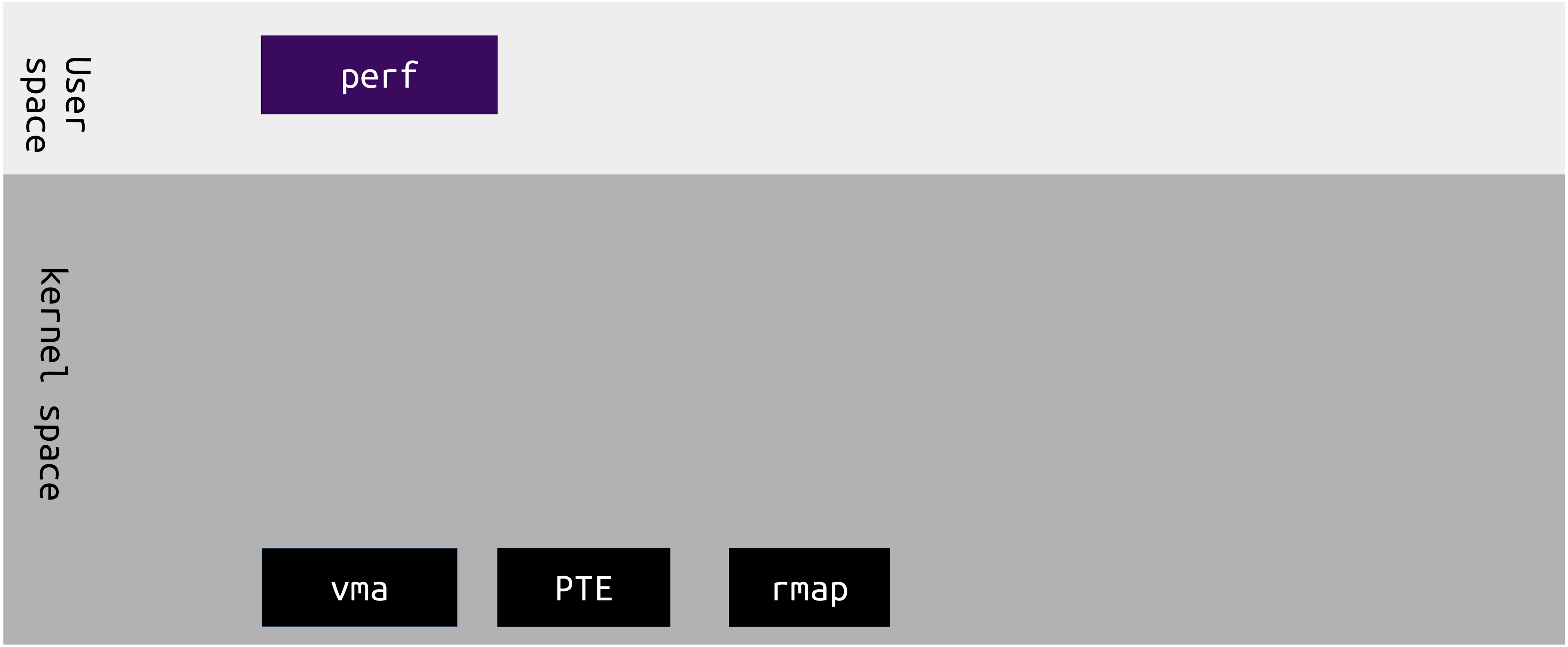
History of DAMON Project

- 2019.03: A prototype research project, DAPTRACE kicked-off
- 2019.09: Renamed into DAMON, Presented at Kernel Summit'2019
- 2019.12: The development resumed in Amazon
- 2020.01: Posted the RFC v1 patchset
- The work was also introduced in several conferences and medias
  - Kernel Summit'19, MIDDLEWARE Industry'19, Phoronix, LWN, Google Linux Kernel Exchange'20, and Kernel Summit'20

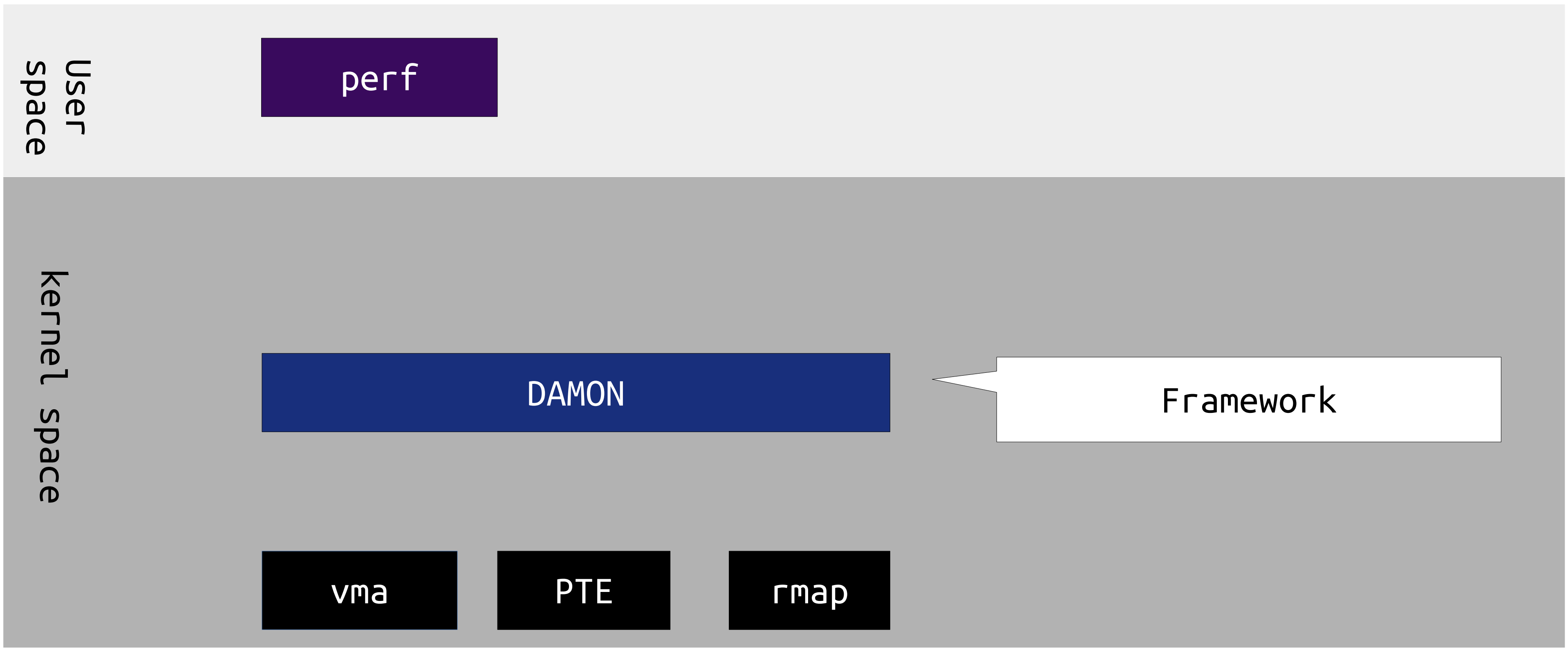
Now: DAMON patchsets series

- V20 DAMON patchset
  - The core of DAMON (framework part)
  - Provide a DAMON primitives for the virtual address spaces
- Two RFC patchsets for future changes are in below order
  - Only to show how DAMON could be evolved
  - Might not ready-to-be-merged level quality
  - RFC v14 of DAMON-based Operation Schemes (DAMOS)
    - Support only virtual addresses
  - RFC v7 of Physical Memory Address Space Support
    - Implement another DAMON primitives for the physical address space
    - Support only mapped LRU pages

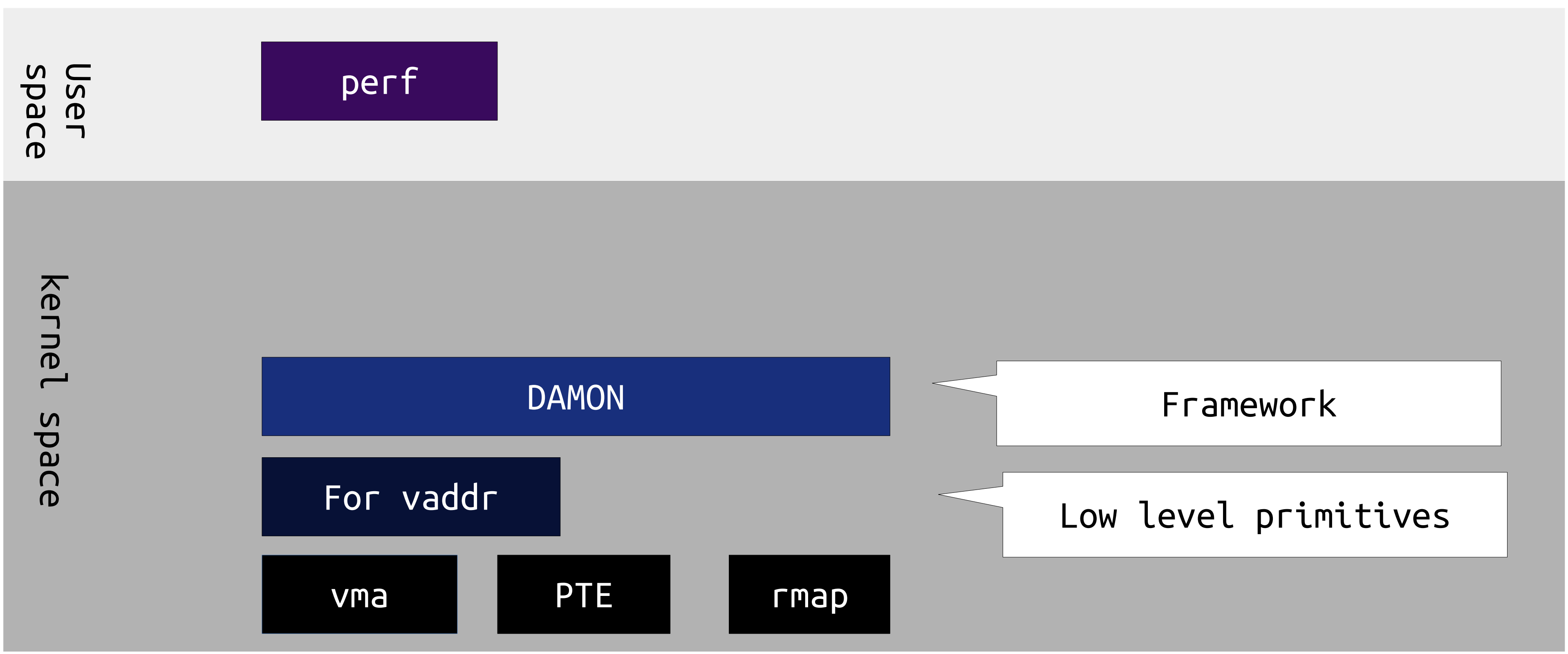
Before DAMON



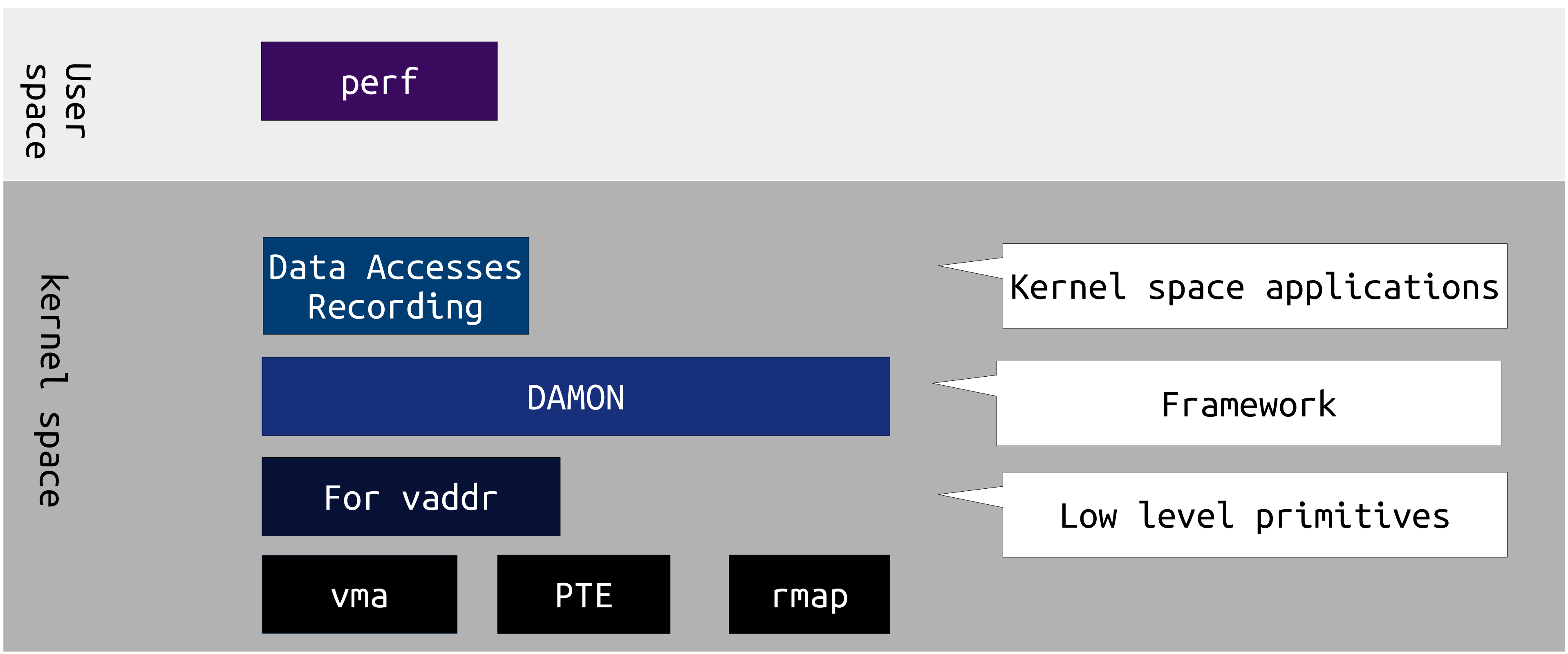
DAMON Patchset 1-4/15



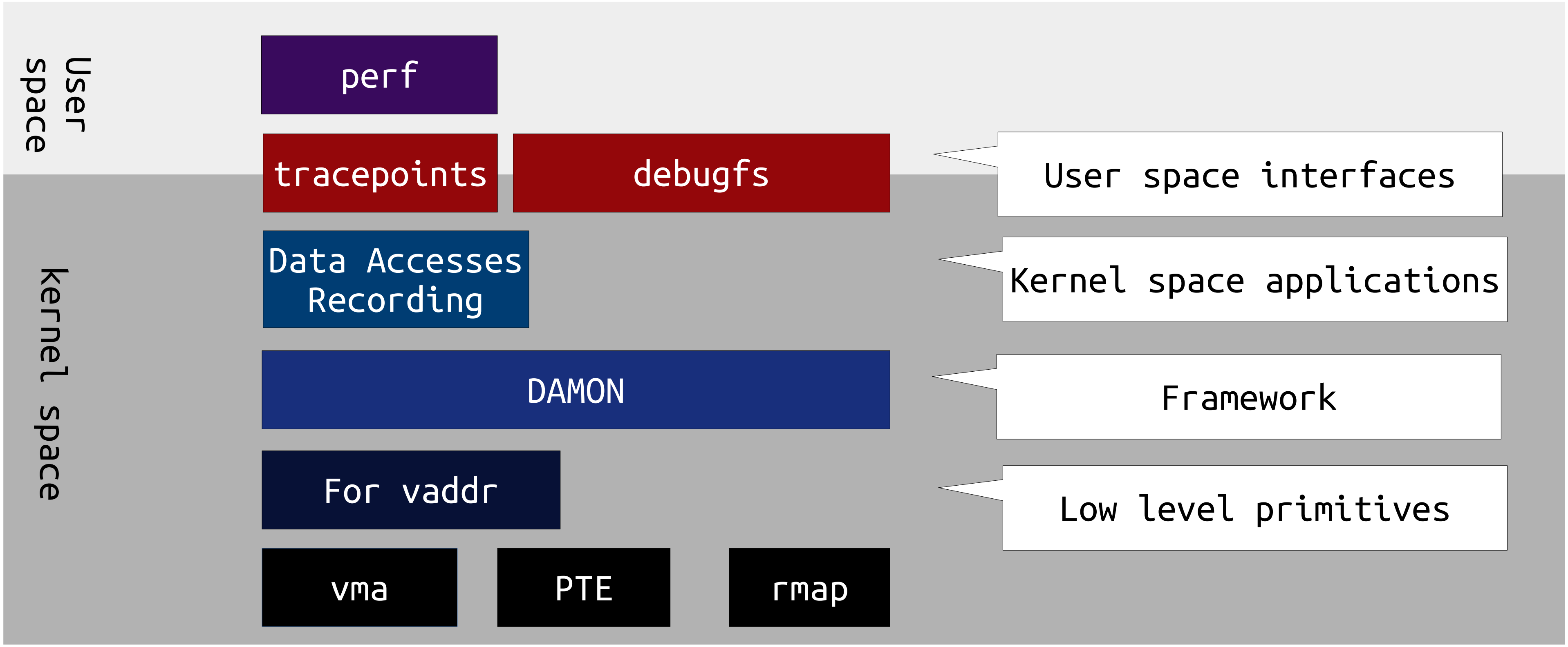
DAMON Patchset 5-6/15



DAMON Patchset 7/15

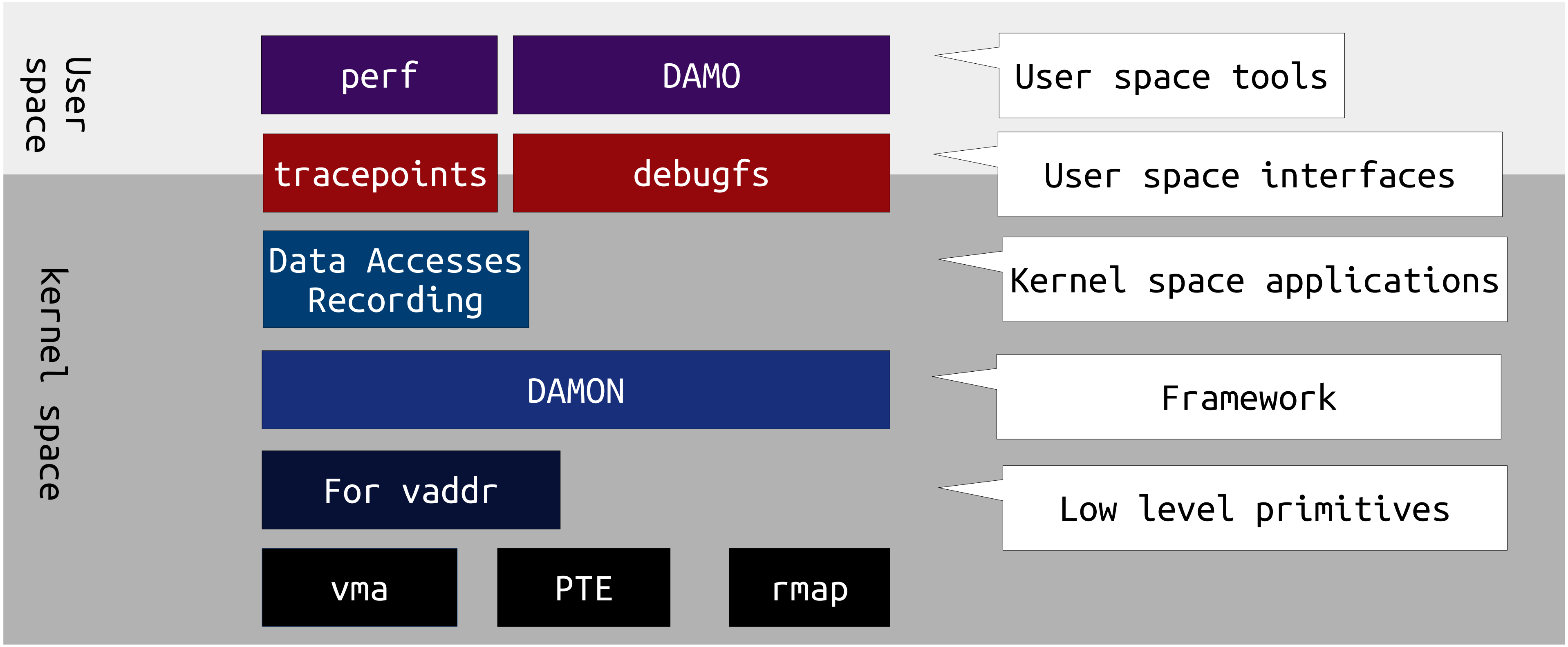


DAMON Patchset 8-10/15

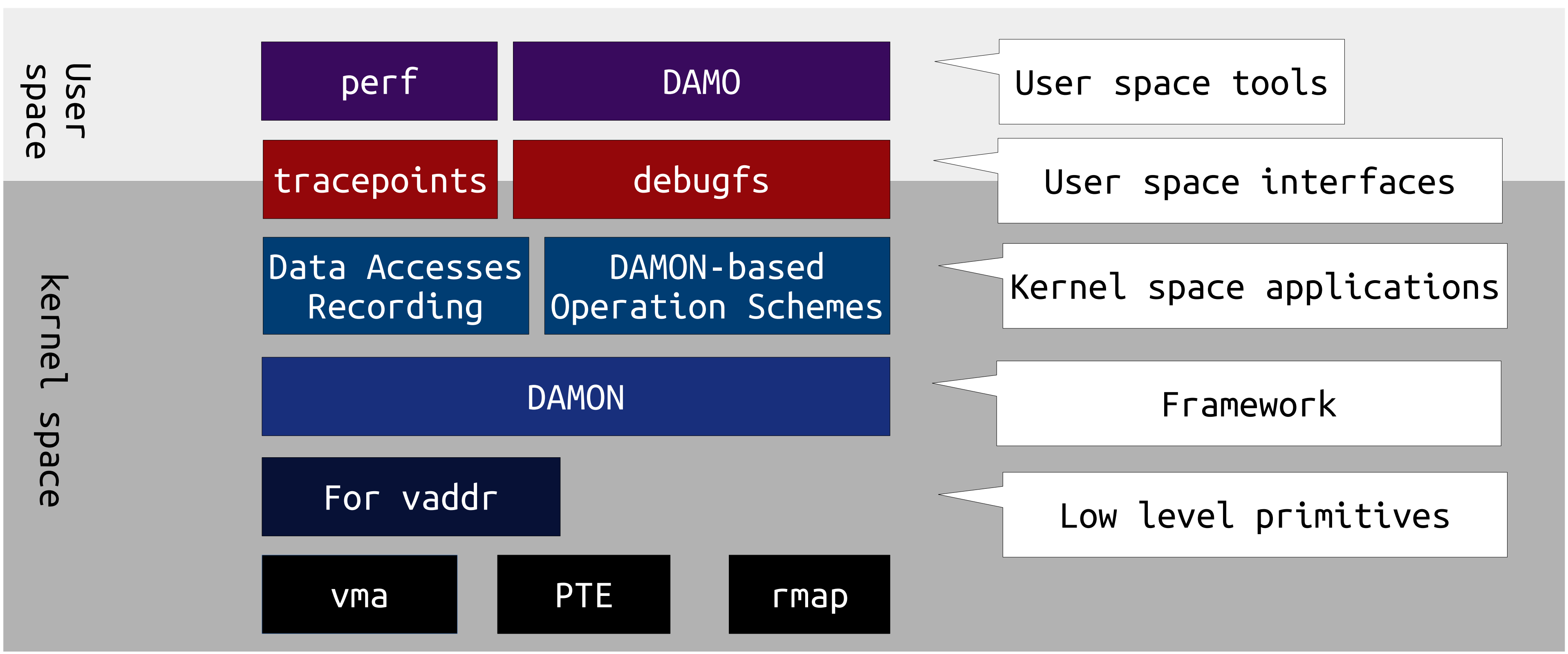


DAMON Patchset 11/15

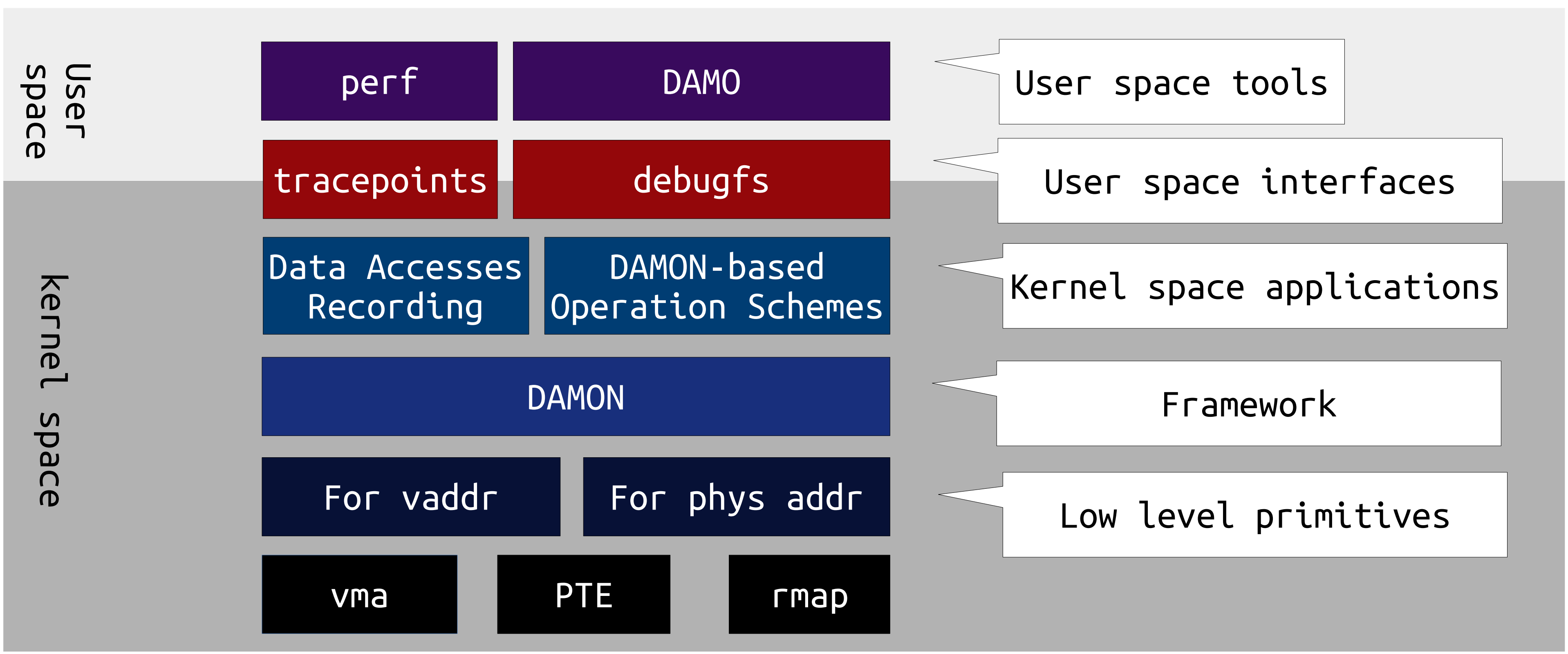
- 12-15 are for documentations and tests



DAMOS Patchset



Physical Address Support Patchset



TODOs

- Make current DAMON patchsets series merged in the mainline
- Support more address spaces
  - Cgroup, cached pages, specific file-backed pages, swap slots, …
  - Physical address support from DAMOS
- Improve the user space interface
  - Multiple contexts, CPU usage charge, …
- Optimize for special use-cases
  - Page granularity monitoring, accessed-or-not monitoring, ...
- DAMON-based MM Optimizations
  - Page reclaim, THP, compaction, NUMA balancing, …

Need Your Opinions

- Are there tasks you want to…
  - Put in the *current* DAMON patchsets before those be merged in,
  - Put in the *future* DAMON patchsets, or
  - Assign higher/lower prioritize?

VOTE: What Task Should Have Highest Priority?

A. Make current DAMON patchsets series merged in the mainline

B. Support more address spaces

- Cgroup, cached pages, specific file-backed pages, swap slots, …
- Physical address support from DAMOS

C. Improve the user space interface

- Multiple contexts, CPU usage charge, …

D. Optimize for special use-cases

- Page granularity monitoring, accessed-or-not monitoring, ...

E. DAMON-based MM Optimizations

- Page reclaim, THP, compaction, NUMA balancing, …

F. Something other? Reply to the patchset, please

Summary

- DAMON is a kernel subsystem providing data access monitoring with
  - Reasonable best-effort accuracy
  - Lightweight overhead and scalable control of it
- Both kernel space and user space could use DAMON for
  - Analysis, MM optimizations, and fun
- Evaluations with 25 realistic workloads say the benefit could be big
- The patchset and two RFC patchsets for future features are available
  - V20 DAMON, RFC v14 DAMOS, RFC v7 Physical address support
  - Review, please!

More Resources: https://damonitor.github.io

- Source Tree: https://github.com/sjp38/linux/tree/damon/next
- Patchsets:
  V20 DAMON, RFC v14 DAMOS, RFC v7 Physical address support
- Document: https://damonitor.github.io/doc/html/next/index.html
- Visualized monitoring results
  - https://damonitor.github.io/test/result/visual/next/rec.heatmap.1.png.html
  - https://damonitor.github.io/test/result/visual/next/rec.wss_sz.png.html
  - https://damonitor.github.io/test/result/visual/next/rec.wss_time.png.html
- Hidden index page: https://damonitor.github.io/_index

https://kids.nationalgeographic.com/content/dam/kids/photos/animals/Birds/A-G/adelie-penguin-jumping-ocean.ngsversion.1396530997321.adapt.1900.1.jpg