Research: Identifying "cold" (rarely accessed) objects and/or memory ranges in the JVM heap ("temperature measurement")

Why do the research?

Moving cold objects **from DRAM to NVM/NVRAM** - high-density lower cost alternative - to **reduce the costs** and **peak memory** usage.

Cold objects can be sequestered (moved to a special separate region), allowing their exclusion from garbage collection, thus reducing the memory footprint and improving cache hit ratios.

IBM Power10 Page access counters

[RFC PATCH v1 0/7] Support arch-specific page aging mechanism

POWER10 supports a 32-bit page access count which is incremented based on page access and decremented based on time decay. The page access count is incremented jirabased on physical address filtering and hence should count access via page table(mmap) and read/write syscall.

This patch series updates multi-gen LRU to use this page access count instead of the page table reference bit to classify a page into a generation. Pages are classified into generation during the sorting phase of reclaim.

[RFC PATCH v1 5/7] powerpc/mm: Add page access count support

https://github.com/kvaneesh/linux/commit/b472e2c8080823bb4114c286270aea3e18ffe221

Hot Cold Affinity engine is a facility provided by POWER10 where each access to a page is counted and the access count value decreased/decayed if not accessed in a time window.

The patch uses HCA engine to provide page access count on POWER10 and uses the same with multi-gen LRU to classify the page to correct LRU generation. This uses a simple page classification mechanism where pages are sampled from the youngest and oldest generation to find the max and min page hotness in the lruvec. This value is later used to sort every page to the right generation. The max and min hotness range is established during aging when new generations are created.

Thermostat: Application-transparent Page Management for Two-tiered Main Memory

Papers: https://web.eecs.umich.edu/~twenisch/papers/asplos17.pdf

Technique

Only requires a target maximum slowdown as input(3% in evaluation) incurred as a result of Thermostat's monitoring and due to accesses to data shifted to slow memory:

- ☐ Periodically sample a fraction of the application footprint and use a page poisoning technique to estimate the access rate to each page with tightly controlled overhead.

- ☐ The estimated page access rate is then used to select a set of pages to place in cold memory, such that their aggregate access rate will not result in slowdown exceeding the target degradation.
- ☐ These cold pages are then continually monitored to detect and rapidly correct any misclassifications or behavior changes.

Each sampling period comprises three stages: (i) split a fraction of huge pages, (ii) poison a fraction of split and accessed 4KB pages, record the access count to 4KB pages to estimate access rate of the huge pages, and (iii) classify pages as hot/cold.

Page sampling. Split a random sample of huge pages (5% in our case) into 4KB pages, and poison only a fraction of these 4KB pages in each sampling interval.

Use hardware-maintained *Accessed* bits to monitor all 512 4KB pages and identify those with a non-zero access rate. Monitor a sample of these pages using a more costly software mechanism to accurately estimate the aggregate access rate of the 2MB page.

Only 0.5% of memory is sampled at any time, which makes the performance overhead due to sampling < 1%.

Page access counting. Current x86 hardware does not support access counting at a per-page granularity.

When a page is sampled for access counting, Thermostat poisons its PTE by setting a reserved bit (bit 51), and then flushes the PTE from the TLB. The next access to the page will incur a hardware page walk (due to the TLB miss) and then trigger a protection fault (due to the poisoned PTE), which is intercepted by BadgerTrap.

BadgerTrap's fault handler unpoisons the page, installs a valid translation in the TLB, and then repoisons the PTE. By counting the number of BadgerTrap faults, we can estimate the number of TLB misses to the page, which is used as a proxy for the number of memory accesses.

Page classification. Sort the sampled huge pages in increasing order of their estimated access rates, and then place the coldest pages in slow memory until the total access rate reaches the target threshold.

Correction of mis-classified pages. Track the number of accesses being made to each cold huge page, using the above approach. In every sampling period we *sort the huge pages in slow memory by their access counts* and their aggregate access count is compared to the target access rate to slow memory. The *most frequently accessed pages are then migrated back* to fast memory until the access rate to the remaining cold pages is below the threshold.

Since the access rate to these pages is slow by design, the performance impact of this monitoring is low. *Identify any mis-classified pages and pages that become hot over time.*

Migration of cold pages to slow memory. NUMA support in KVM guests already exists in Linux and can be used via libvirt

Details

Classifies both 4KB and 2MB pages as hot or cold. Implemented Thermostat in Linux kernel version 4.5.

	Performance gain
Aerospike	6%
Cassandra	13%
In-memory Analytics	8%
MySQL-TPCC	8%
Redis	30%
Web-search	No difference

Table 1. Throughput gain from 2MB huge pages under virtualization relative to 4KB pages on both host and guest.

Can be controlled at runtime via the Linux memory **control group (cgroup)** mechanism. All processes in the same **cgroup** share Thermostat parameters, such as the sampling period and maximum tolerable slowdown.

To bound the performance impact of access rate monitoring, only a small fraction of the application footprint may be monitored at any time. However, sampling only a small fraction of the application footprint leads to a policy that adapts only slowly to changes in memory behavior.

Uses **THP(Transparent Huge Pages)** - a feature in the Linux kernel (available since version 2.6.38) that automates the use of huge pages, attempts to allocate huge pages dynamically and fall back to the regular 4KB pages when the huge page allocation fails.

The **perf** utility, also known as **perf_events** or Performance Counters for Linux (PCL), is a powerful tool for performance profiling, tracing, and monitoring in Linux. For pages we identify as cold, the TLB miss rate is typically higher (but always within a factor of two) of the last-level cache miss rate measured without BadgerTrap,

Kstaled framework. To distinguish cold pages from frequently accessed hot pages, existing mechanisms exploit the *Accessed* bit in the PTE (set by the hardware each time the PTE is accessed by a page walk).

Single accessed bit per page is insufficient to distinguish hot and cold 2MB huge pages with sufficiently low overhead. To detect a page access, **kstaled** must clear the *Accessed* bit and flush the corresponding TLB entry. However, distinguishing hot and cold pages requires monitoring (and hence, clearing) accessed bits at high frequency, resulting in unacceptable slowdowns.

Approach assumes that the *number of TLB misses and cache misses to a 4KB page are similar*. For hot pages, this assertion does not hold, but it doesn't matter. For cold pages, nearly all accesses incur both TLB and cache misses as there is no temporal locality for such accesses, and, therefore, tracking TLB misses is sufficient to estimate the page access rate.

Set a tolerable slowdown of 3% throughout our evaluation, since a higher slowdown may lead to an overall cost increase due to higher required CPU provisioning (which is more expensive than memory).

Results

Very promising…

Evaluate Thermostat with applications from Google's Perfkit Benchmarker and the Cloudsuite benchmarks - MySQL-TPCC, NoSQL(Aerospike, Cassandra, Redis), In-mem analytics, Web-search running under KVM virtualization.

5% of huge pages sampled in every scan interval of 30s and at most 50 4KB pages poisoned for a sampled huge page.

Thermostat migrates up to 50% of application footprint to slow memory while limiting performance degradation to 3%, thereby reducing memory cost up to 30%.

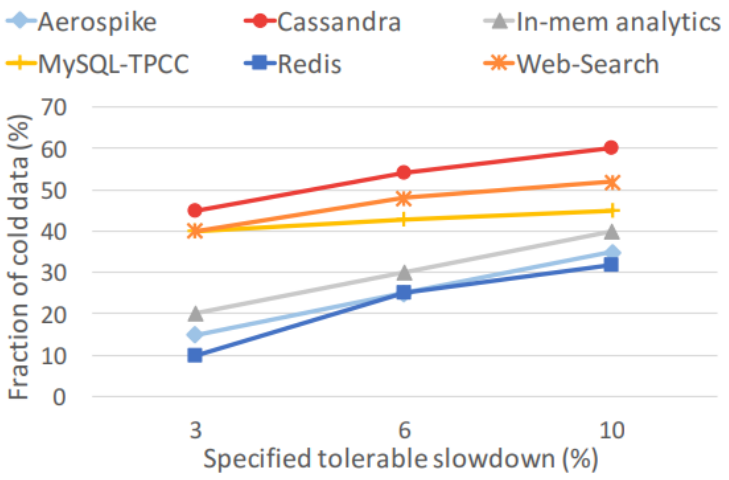

Figure 11. Amount of cold data in identified at run time varying with the specified tolerable slowdown. All the benchmarks meet their performance targets (not shown in the figure) while placing cold data in the slow memory.

Further work

Fault on LLC miss. Software-only page access counting mechanism has two sources of inaccuracy - TLB misses instead of LLC, measurement process throttles accesses to the poisoned pages.

TLB misses, which represent a subset of LLC misses, can be observed at the OS level through the reserved bits in the PTE without requiring extensive hardware support.

PEBS based access counting. PEBS (Precise Event Based Sampling) subsystem in the x86 architecture can be extended to record page access information. In the current PEBS implementation, a PEBS record is stored in a pre-defined memory area on samples of specific events (LLC misses is one of them). When the area fills up, an interrupt is raised, which the kernel can then service and inspect the instructions that led to those events. Default sampling value is too low in linux kernel, default PEBS memory is too low to store the whole CPU state.

Materials to check

Exposing kernel functionality to the user space

AutoNUMA

AutoNUMA is an automatic placement/migration mechanism for colocating processes and their memory on the same NUMA node to optimize memory access latency.
AutoNUMA relies on CPU-page access affinity to make placement decisions.
Works in cache coherent NUMA multi-processors, wherein no software fault is required.
A migration mechanism seeks to shuffle pages between tiers to maximize fast-memory accesses.

J. Corbet. AutoNUMA: the other approach to NUMA scheduling. http://lwn.net/Articles/488709/, 2012. [Online; accessed 9-May-2016].

GC JVM

GC prerequisites

IBM OpenJ9 is a high-performance, open-source Java virtual machine (JVM) developed by IBM - https://github.com/eclipse-openj9/openj9

Gencon and Balanced are **region-based policies** that are used in IBM's OpenJ9 virtual machine to manage the memory of Java applications.

GC algorithms evaluation:

- Throughput: amount of work performed by the application threads;
- GC overhead: the measurement of processor time used by the garbage collector;
- Pause time: the amount of time an application is stopped during GC;
- Frequency of garbage collection;
- Footprint: maximum heap size;
- Promptness: time between death of an object and when the memory becomes available;
- Completeness: all garbage has to be reclaimed eventually.

Additional

- Fragmentation: This metric measures the degree to which memory is fragmented after GC. Higher fragmentation can lead to performance issues as the application needs to search for contiguous blocks of memory.
- Scalability: This metric measures how well the GC algorithm scales as the size of the application and the number of threads increase.

Some popular benchmarks for evaluating GC algorithms include **SPECjvm2008**, **DaCapo**, **CloudGC** and JBoss Enterprise Application Platform (**EAP**) Benchmarks.

Most garbage collection policies fit into at least one of four categories: mark-sweep, mark-compact, copying and reference counting.

Stack walking(JVM)

Papers: Cold object identification in the Java virtual machine

Technique

VERY COMPLEX SOLUTION.
Activity sampling daemon polls/triggers Mutator Threads, updates reference counters r for regions, marks bits in activity map for pinned/cold regions
Mutator threads walkdown from topmost until a frame that has not been active, collect references, hold previous + current trace
Region pinning/unpinning - pin a collection of tenured regions that contains as much of the mutator's active working set as possible, selective vs nonselective pinning
Cold regions can be found within the pinned regions that are receiving few references after fixed Tcold threshold. Included in the copy forward collection set for the next partial GC cycle if … . Then collectible cold objects are copied into the next available cold region while all other objects are copied into other unpinned regions.

A number of tenured regions are first pinned so that they are excluded from partial GC collection sets cycles. This ensures that objects contained within pinned regions maintain a fixed location within the mutator address space. Pinned regions are also instrumented with activity maps to record which objects have been sampled from mutator stacks. Cold objects can be identified within a pinned region, after a preset amount of time Tcold has elapsed since the most recent setting (0 -> 1) of an activity bit, by subtracting the activity map from the mark map.

2.2.1. Pinned region selection. The region pinning framework partitions the regions of the balanced GC heap into four collections:

(1) young regions (age < 24),
(2) unpinned regions (age 24, not pinned),
(3) pinned regions (age 24, pinned), and
(4) cold regions (the cold area).

Unpinned regions are considered for pinning at the end of each partial GC cycle. They are selectable if they have an allocation density d (ratio of allocated bytes to region size R) exceeding a preset threshold D_{hi}. Additionally, the total size of potentially collectible cold objects contained in the region must be greater than $0.01R$. Selectable regions are ranked at the end of each partial GC cycle according to the region pinning metric value P that reflects the volume of activity in each region:

$$P = mma(r) * d \tag{1}$$

where r is the number of mutator references to contained objects since the end of the previous partial GC cycle and is a measure of the object activity in the region. The $mma(r)$ is the modified moving average of r with a smoothing factor of 0.875:

$$mma(r_0) = 0; mma(r_n) = \frac{(7 * mma(r_{n-1}) + r_n)}{8}, n > 0 \tag{2}$$

The maximum number of regions that may be pinned at any time is determined by a preset parameter P_{max}. At the end of every partial GC cycle, if the number of currently pinned regions n is less than P_{max}, up to $P_{max} - n$ selectable regions may be pinned.

Details

Targets objects within the Java heap that are persistent but seldom if ever referenced by the application. Examples include strings, such as error messages, and collections of value objects that are preloaded for fast access.

Balanced GC - management of cold objects is most relevant in the context of long-running applications with large heap requirements. For that reason, the balanced gc (GC) framework was selected as a basis for investigating cold object management.

In order to preclude the need to traverse cold regions during mark/sweep actions, only arrays of primitive data (e.g., char[]) and leaf objects (objects with no reference-valued fields) are considered as collectible cold objects. This constraint, in conjunction with the leaf object optimization feature available in the IBM Java virtual machine, ensures that objects within cold regions can be correctly marked but not touched by the marking scheme.

Results

Measure number of cold objects/bytes, FalseInactivity(use access barrier), ConvergenceTime (from pinned before being determined as cold).

Not good.
The runtime overhead for walking mutator stacks and maintaining pinned activity maps offset any gains that accrued from establishing the cold area and marshaling cold objects out of resident memory, but it may be possible to reduce overhead by optimization. This approach can miss potential hot objects during sampling intervals and

is effective with leaf objects only. In this paper, only primitive arrays and leaf objects are considered as cold objects

Most of the ssd-stack benchmarking runs yielded a few tens of megabytes in the cold area and consumed 4–6% of the available CPU bandwidth, which is a relatively high price to pay for the amount of cold data collected.

For example, the number of collectible pinned regions in access barrier is five times larger than in stack-based solution; the number of cold objects in access barrier is 11.78 times larger than that in stack-based solution; and the size of cold objects is 10.42 times larger in access barrier than that in a stack-based solution.
The FalseInactivity ratio is 1.62%, which is quite low.

Page Protection with Profiling(JVM)

Papers: Cold Object Identification and Segregation using Page Protection and Profiling, COLD OBJECT IDENTIFICATION AND SEGREGATION VIA APPLICATION PROFILING

Technique

The PGC consists of either a copy-forward or mark-compact collection. It allows copying an individual object into a specific region, e.g., older objects are moved to the tenured region. The temperature data is used during this phase to further segregate objects into hot or cold regions, which consists of the following steps.

• The profiled data is read into an in-memory hash table during JVM initialization.
• A page-fault handler is registered to use OS paging as an access barrier.
• The page protection technique identifies hot pages.
• A single page fault can mark all the objects in a page as hot. The profiled data is used to refine the object classification as hot or cold.

Details

• Only non-Eden regions are protected
• Profiled data further guides the final destination for the object**.**

A balanced policy copy-forward operation is modified to handle object segregation as part of this research.
The cold allocation context provides an easy to manage list of cold regions, backed by a separate memory device, e.g., NVRAM or an SSD partition. References to “Persistent Memory Storage of Cold Regions in the OpenJ9 Java Virtual Machine”.

Temperature is calculated by summing up the access frequencies for each incoming reference. Similarly, the sum of frequencies for the outgoing references gives the popularity measure. The temperature data is summarized as class-level statistics.

These statistics are gathered during the pre-collect phase of a GC cycle, i.e., when the application threads are stopped.

Results

Measured runtime overhead only, there are cases where runtime is better.

There is overhead as expected - long-running applications with better nursery survival rate will benefit more from this solution.

For one of the test runs, Investigation of the GC logs shows that the heap was unable to perform a copy-forward operation, because the heap was too full. Hence not enough segregation was performed for the H2 benchmark, leading to poor runtime with page-protection. Incorporating the segregation logic for mark-compact operations should address this issue.

Assumption that all objects of a given class have the same temperature is not correct. Therefore, the classification is not optimal. The profiled data should be re-evaluated at runtime. The IBM Z14 platform supports a guarded storage facility (GSF), which allows applications to guard a memory-region. A hardware read barrier is provided for such guarded regions to give better granularity over page protection.

Stack walking can be used to update the temperature data at runtime.

Page Marking/Protection(JVM)

Papers: [Persistent Memory Storage of Cold Regions in the OpenJ9 Java Virtual Machine]

Technique

A bitmap is created, with one bit per operating system page in the heap memory. A bit set to 0 in the bitmap means that all objects in the page are cold. After a garbage collection is performed all bits in the bitmap are set to 0, and all pages in the heap protected from memory access using the mprotect Linux system call, causing all heap accesses to generate segmentation faults in the processor.

Using the sigaction Linux utility, a callback is registered to handle segmentation faults. This call back checks if the faulting address is a heap address, and if it is, turns the protection off for that page and marks the page as hot by setting its corresponding bit in the bitmap to 1. At garbage collection time, objects in pages marks as cold are known to be cold and all other objects must be assumed hot.

A single read/write access marks all the objects in the page as hot objects, creating false positives.

Details

Used BalancedGC, as the scheme already supported many features that were identified as useful for temperature-based object segregation, including region ageing, per-object

copying and an allocation context that prevented accidental misuse of NVM. Modifying BalancedGC required modifying the OMR memory model.

The copy forward operations in BalancedGC select an allocation context that objects are copied into when a region is collected. The source of the selection was modified to support contextual selection of the allocation context when copying an object.

Result

Measured runtime overhead for different heap sizes, object allocation throughput, heap size over time, average response time over time.

Kind of big runtime overhead in most cases. Very few configurations with a large number of long lived objects and large heap sizes(3+ Gb) showed improvement…

Where this technique did not perform better, the cause was identified as overhead in the scheme, a failing of the scheme to deal with more sophisticated loads or a failure to trigger the copying mechanism.

Where there was a performance improvement, the cause was identified as reduction of time between garbage collections, indicating an improvement in object locality. This was identified by comparing the applications' running time to the time spent performing GCs.

JVM(Access-barriers)

Papers: no direct papers, though "Page Protection and Profiling" use it for cold object statistics and refer to it as slow, "Stack walking" used as a verification tool.

Technique

An access barrier allows the JVM to intercept any read or write operation to an object reference. These access barriers can be configured to record the last access time for an object. Any object that remains inactive for a certain time threshold should be treated as cold. The GC cycle being a stop-the-world operation offers the best opportunity to analyze the object access timestamps.

Frequently executed methods are compiled to native instructions by the JIT compiler, which takes away the majority of access barriers, limiting the usefulness of this approach i.e., it works only in JIT-off mode.

An additional method call on every object access is clearly not performant and hence this approach is only useful as a verification tool.

Details

They probably messed with the access barriers implementation because of "JIT-off mode" and "additional method call", or they tried, and it appeared to be inefficient.

Card Tables(JVM)

Papers: no papers

In addition to performance overhead, page protection is too coarse-grained to avoid false positives. A card represents a contiguous chunk of memory in the heap, which is much smaller than a page size.

Though, it improves the granularity over the page protection scheme, only write operations are recorded. Real life applications exhibit significantly higher read operations in comparison to writes, which limits the usefulness of card table.

Offline&Online profiling (HotSpot JVM, Power consumption)

Papers: Memory Power Consumption Minimization Through Cold Object Segregation

Develop sampling and profiling-based analyses and modify the code generator in the HotSpot VM to understand object usage patterns and automatically determine and control the placement of hot and cold objects in a partitioned VM heap.

Technique

VM consults the available usage information at each object allocation point to determine which color to assign the new object. Periodically, perhaps at every garbage collection cycle, the VM may also re-assign colors for live objects depending on the predicted access patterns and / or program phase behavior.

Offline Profiling. For offline profiling, they capture information about the usage of objects related to when they were allocated and label those allocation points as hot or cold depending on their access rates.

Instrument the HotSpot bytecode interpreter to construct a hash table relating objects and their usage activity to their allocation points in the interpreted code. The hash table is indexed by program allocation points (identified by method and bytecode index) and values are a list of records describing the size and access count of each object created at that point. During the run, whenever the VM interprets a _new instruction, it creates a record. Whenever an object is accessed using one of the object reference bytecodes (getfield, putfield, etc.), the VM increments its access counter.

0/1 knapsack optimization problem. Each allocation point is an item, which has a value equal to the total size of the objects created at that point, and a weight equal to the sum of the access counts for those objects. Use some maximum number of object accesses is the capacity of the knapsack.

Online Profiling. For online profiling, they record information about what classes are the targets and sources of load and store operations for each compiled method and periodically check the stack during runtime to assess what methods are being called. This information is used to assess whether a class is likely to be hot, and all objects of hot classes are considered hot.

The HotSpot compiler records the list of classes with instances that might be accessed by the compiled method. If a class with an instance being the source/target of a

load/store instruction in the compiled method, it is added to a list associated with the compiled method.

A separate thread, invoked by a timer-driven signal, periodically examines the call stacks associated with each application thread. For each method at the top of a call stack, a counter associated with the method is set to indicate that the method is hot. Counters associated with all of the classes on the method's ClassAccessList are set to indicate that instances of this class are likely to be hot. To cool down, periodically decrement the counters associated with the hot methods and classes - non-active methods/classes will eventually become cold.

If either counter indicates that the new object is likely to be hot, we allocate the object to the hot space. At each garbage collection cycle, the surviving objects' colors are reassigned according to the hotness/coldness of their classes.

Details

Focus on JEDEC-style DDR* memory subsystems

Application provides a usage map(usage patterns through colors for the virtual pages) to the OS, and the OS consults this usage map in selecting an appropriate physical memory scheduling strategy for those virtual page

Divide the application's heap into two regions: one for objects that are accessed relatively frequently (i.e. hot objects), and another for objects that are relatively cold. Colors are applied to each space during initialization and whenever region boundaries change due to heap resizing.

For all of our experiments, we configure HotSpot to use a static heap size to avoid re-computing colored space boundaries during heap resizing.

Results

They didn't target cold head size directly.

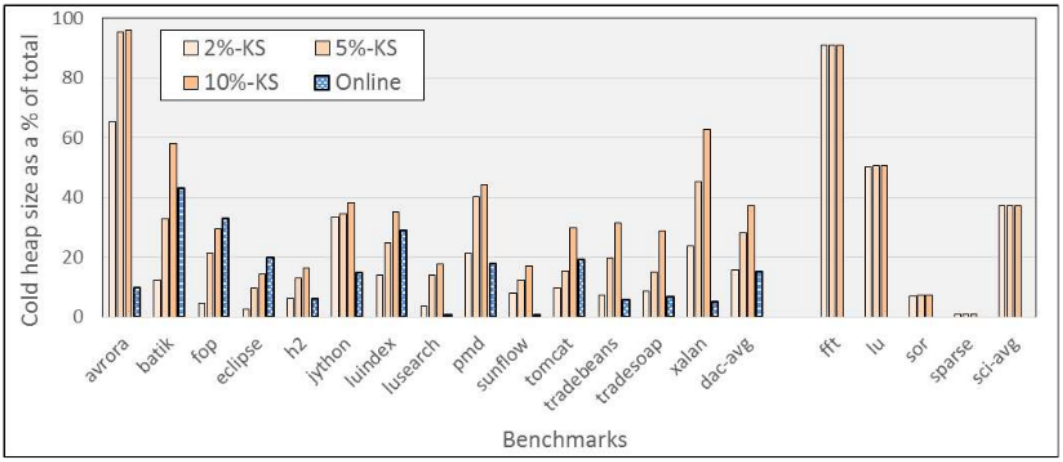

Figure 6. Cold heap size as a percentage of total size with different object partitioning schemes.

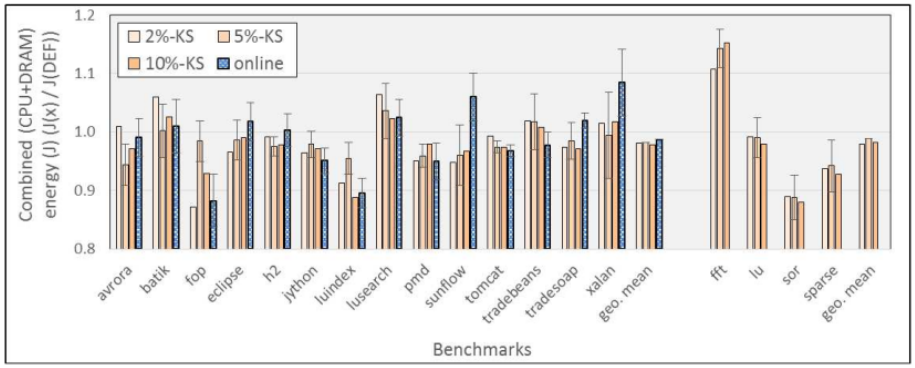

Figure 14. Combined (CPU+DRAM) energy consumed with each colored configuration compared to default.

Indirectly related approaches(JVM)

Field Reordering and Object Splitting

Eimouri investigated static profiling of field access rates at the class level in an attempt to optimize the performance of runtimes. They used static profiling, changed the layout and access mechanism of classes, and focused more on cache performance.

To perform the profiling, access barriers are used to trace different accesses to different fields of various classes. We need information like class name, the number of non-static fields and their names, types and modifiers. In addition, we need to know the access frequency (hotness) of each non-static field. To keep this information during the execution of the application, a hash table is used.

Useful materials

Power10 performance counters or page access counters, Thermal stats, arrange a meeting with IBM to get code