Read-only Analysis For Java Objects

Motivation

The background of heterogeneous memory system is introduced in the previous project proposal [1]. The motivation of this analysis is to try to mitigate the “data migrating” overhead in a heterogeneous memory system. The on-chip memory has high-bandwidth but also a limited space compared with the commodity memory. The hot objects will be placed in the high-bandwidth memory according the guide from the upper level application. When the the high-bandwidth memory gets full, some objects need to evicted to make space for new objects. The reason why we study the read-only objects lies in the advantage of avoiding copying them back to the off-chip memory if there are old copies of them.

Read-only data optimization is an old topic, but we are applying it to the new memory technique. Specifically, when the high-bandwidth memory is full, an on-demand data migration to the off-package memory would require the OS to perform the following steps:

1. Interrupts and suspends all the threads.
2. Copies the evicted objects back to the main memory.
3. Updates the page table entry.
4. Flushes TLB cache.

All of these operations come with significant overheads [2]. However, empirically, we believe that a significant portion of application data is only written shortly after it is allocated, but only read frequently thereafter. Then, if the system decides to promote some read-only or read-mostly data to the high performance tier, it will also keep a copy of the data in the lower tier. In this way, as long as the data is never modified in the upper tier, the system can migrate it back to the lower tier by simply adjusting the corresponding PTEs to point back to the original copy, but without having to copy the data to the lower tier.

Furthermore, the optimization above also enhances another TLB optimization when the eviction happens proactively instead of on-demand. When the proactive eviction strategy finds some read-only objects in the higher tier become cold, it adjusts the corresponding PTE and reclaim the copy in the high performance tier even when there are no new hot objects needing to enter the higher tier. And because a TLB flush “naturally” happens between context switching, the PTEs that are updated earlier for free.

Related Work

There are many different perspectives to measure the percentage of read-only objects. Some work [7] measures the number of accesses to read-only objects and the total memory accesses. But under the context of memory system, measuring the size ratio of read-only objects instead of access ratio is more relevant.
The effectiveness of the optimizations depends on the proportion of the read-only or read-mostly data in the application. Therefore, to demonstrate the potential of this approach, we conducted a set of experiments with DaCapo [3] benchmarks and SPEC CPU2006 [4] benchmarks. The instrumentation approach and the experimental results will be shown in he following sections.

Methodology

We hold the assumption that objects allocated at a same allocation site tend to have similar access patterns, so as an initial investigation, the analysis is performed at the allocation site level, namely treating all objects the same if they are allocated under the same context.

For the DaCapo benchmark, we instrument HotSpot 9 JVM [5] interpreter to record object events with -Xint on. In the first phase, we collect the number of objects allocated at a certain allocation site as well as the size and references to objects allocated before each GC happens as the applications run in the interpreter mode. The reference is further differentiated as read or write. All the object accesses are recorded in the output file even they are reclaimed later by the garbage collector. In the second phase, we process the output file produced by the first phase and calculate the percentage of read-only objects, 95% read objects, 90% read objects and 80% read objects. The way we define a 95% objects is that the reads to that object contribute to more than 90% of the total accesses, and the similar definition goes for 90% and 80% read objects.

We applied the similar instrumentation and post-processing for SPEC CPU2006 benchmark, except that we use a Pin [6] tool to intercept and record the object access. Object in this context means the data dynamically allocated in the heap.

Experimental Results

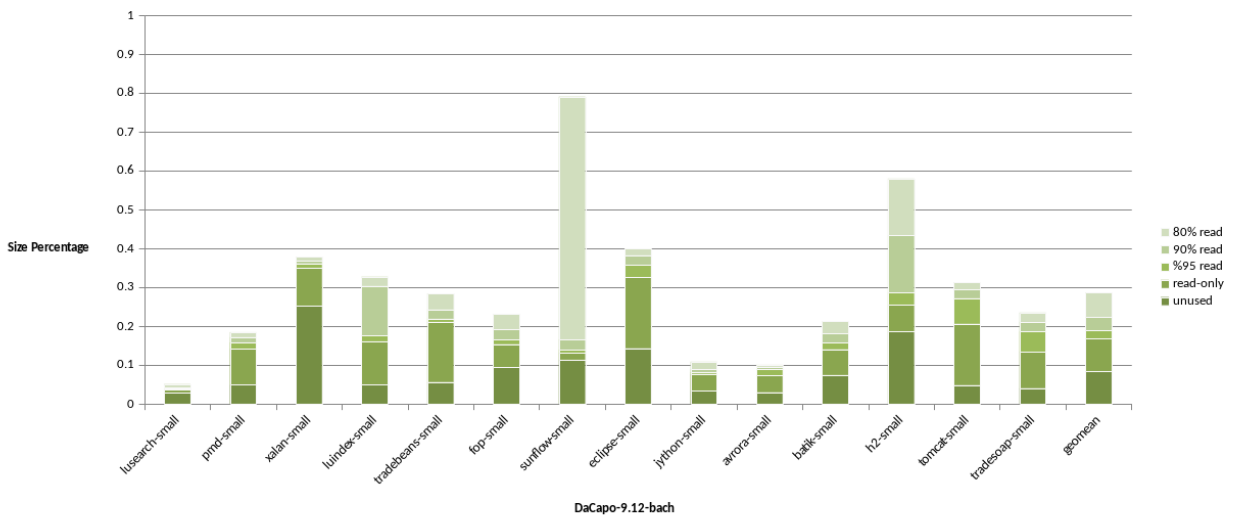

Figure 1: Ratio of read-only and read-mostly data to the total number of bytes allocated for the small DaCapo suite of Java benchmarks.

Figure 1 shows the percentage of the objects in small sized DaCapo benchmarks that are read-only or read-most in terms of space they take. Overall 16.7% of the application objects are read-only, including unused objects. Axlan-small has the highest read-only ratio, which is 35%. Figure 2 shows the results for default-sized DaCapo benchmarks, which is very close to the results of small suite.

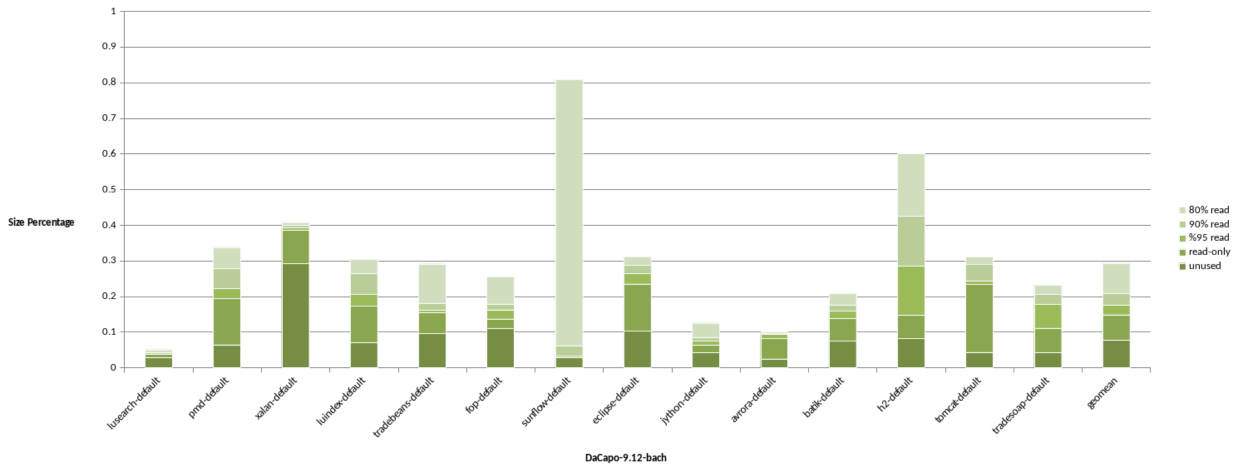

Figure 2: Ratio of read-only and read-mostly data to the total number of bytes allocated for the default DaCapo suite of Java benchmarks.

Figure 3 shows the result for CPU2006 benchmarks. It has a greater variation compared with the DaCapo benchmarks. Overall only 4.25% of the objects are read-only. Note that all objects in 429.mcf-ref are 95% read, signaling a possible "write once read forever" access pattern, which encourages us to conduct further explorations.

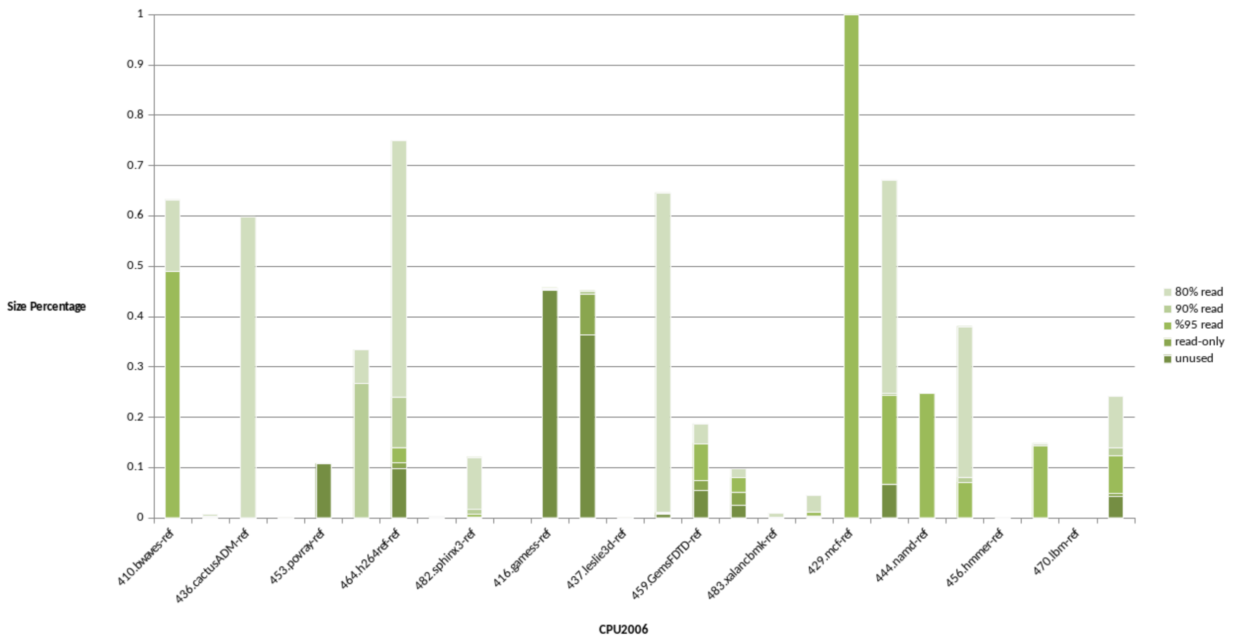

Figure 3: Ratio of read-only and read-mostly data to the total number of bytes allocated for SPEC cpu2006 benchmarks.

Unused Analysis

Because a significant portion of the read-only objects are unused, we continue to conduct an unused analysis. The new experiment has been carried out on the DaCapo benchmarks. We instrument the HotSpot 9 VM the same way as we did in the read-only analysis. On top of that, to verify the assumption that a lot of objects are only used shortly after it is created, we sample and inspect the heap every 100 ms. Then in the second phase, we calculate the portion of the objects that are “never used”, “only used in the first interval after creation”, “only used in the first two intervals” respectively. This is our first step of exploring the access pattern of heap objects.

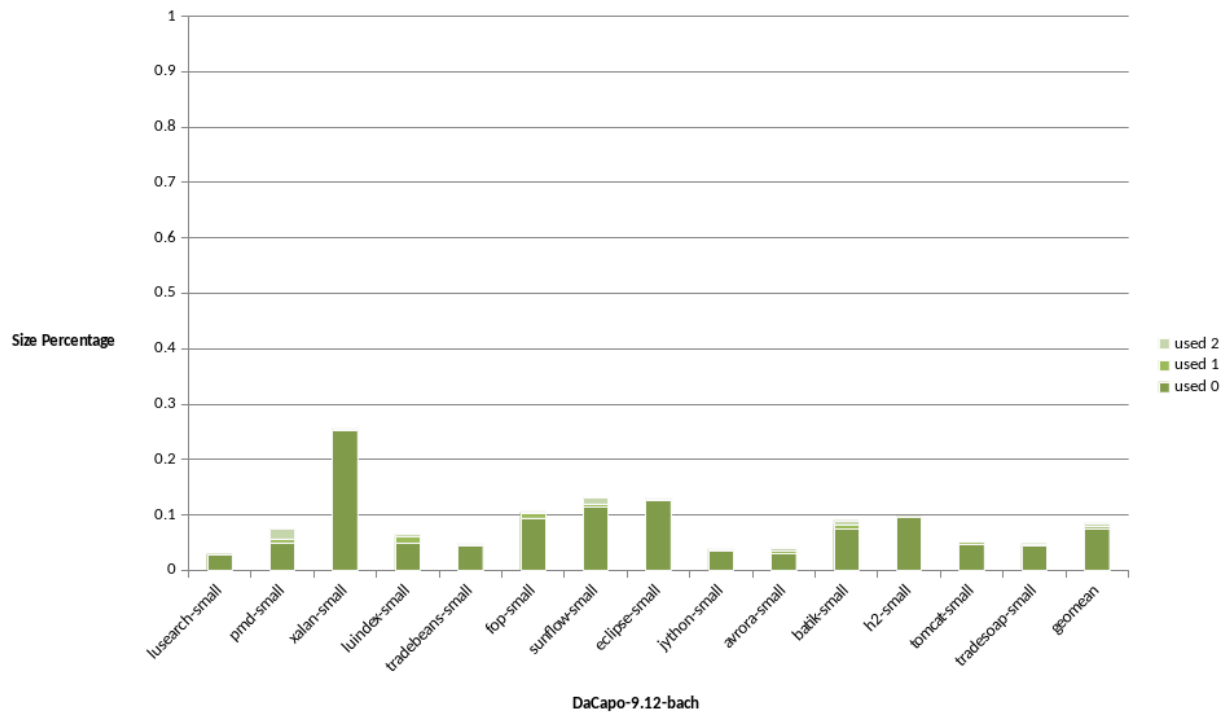

Figure 4: Ratio of data that are only used shortly after creation to the total number of bytes allocated for small DaCapo benchmarks. “used 1” means “only used in the first interval”, etc.

Figure 4 shows that almost no objects are used only in the first interval or the first two intervals. The default sized DaCapo benchmarks also show the same result. This result is highly counter-intuitive so we are currently conducting a new set of experiments that does the same analysis in an object level instead of an allocation site level. Because two different objects from a same allocation site could be used in different intervals, we expect the portion of the objects that are only used in the first interval to be higher in the new experiments.

Interval Analysis

In this section, we show the percentage of objects that are only accessed in the first 100ms, 200ms, 300ms after their creation. We measure the percentage both in terms of number of objects and in terms of size of objects. Unlike the previous section, we measure the read-onlyness of each individual objects and then group them into allocation sites.

Figure 5 shows that 95% of the objects are only used in the first 100ms after creation. 95% is the ratio of the number of all objects to the number of read-only objects. Figure 6 shows a very different results with defining the read-only ratio as the size of read-only objects to the size of all objects. The two figures together imply that most read-only objects are small objects and do not account for much memory usage.

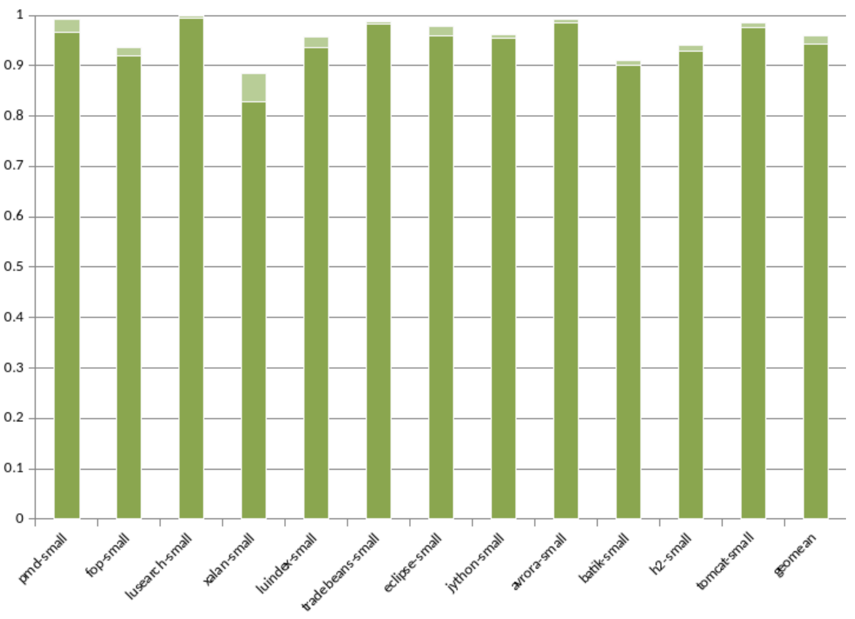

Figure 5: # of read-only objects / # of all objects for small DaCapo benchmarks.

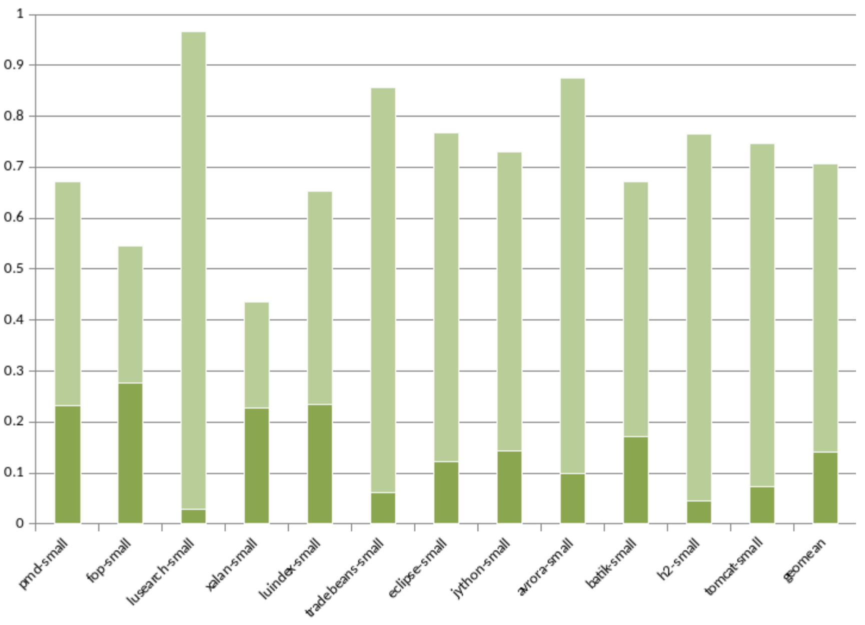

Figure 6: # of bytes of read-only objects / # of bytes of all objects for small DaCapo benchmarks.

Conclusion

We show that about 16.7% objects in DaCapo benchmarks are read-only, 8% of which are unused. Extending the threshold to 95%, 90% and 80% read have a respectively 2.7%, 3.1%, 8.5% increase in the percentage. For CPU2006 benchmarks, 4.25% of the objects are read-only, 4.2% of which are unused. Extending the threshold to 95%, 90% and 80% read have greater impact on the results, which are 7.6%, 1.5%, 10% increase respectively.

In addition, 95% of all objects are only used within the first 100ms after their creation, but they only account for 14% of the allocated memory.

Among the long-lived objects, some of them may be only written briefly and read-only afterwards. We are not sure what advantage that might bring us, but in the future work, we will get some statistics for these objects.

Reference

[1] The previous project proposal
[2] M.R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G.H. Loh. Heterogeneous
memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In
High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on,
pages 126–136, Feb 2015.
[3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley,
Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel,
Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, B. Moss, Aashish Phansalkar, Darko Ste-
fanovi´c, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo bench-
marks: Java benchmarking development and analysis. In Proceedings of the 21st annual ACM SIG-
PLAN conference on Object-oriented programming systems, languages, and applications, OOPSLA
’06, pages 169–190, New York, NY, USA, 2006. ACM.
[4] John L Henning. Spec cpu2006 benchmark descriptions. ACM SIGARCH Computer Architecture
News, 34(4):1–17, 2006.
[5] http://openjdk.java.net/projects/jdk9/
[6] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

[7] Specific read only data management for memory hierarchy optimization