Efficient Management for Hybrid Memory in Managed Language Runtime

Chenxi Wang, Ting Cao, John Zigman, Fang Lv, Yunquan Zhang, Xiaobing Feng

Efficient Management for Hybrid Memory in Managed Language Runtime

Chenxi Wang[1,2], Ting Cao[1], John Zigman[3], Fang Lv[1,4], Yunquan Zhang[1], and Xiaobing Feng[1]

[1] SKL of Computer Architecture, Institute of Computing Technology, CAS, China
{wangchenxi, caoting, flv, zyq, fxb}@ict.ac.cn
[2] University of Chinese Academy of Sciences, China
[3] Australia Centre for Field Robotics, AMME, The University of Sydney, Australia
john.zigman@sydney.edu.au
[4] State Key Laboratory of Mathematical Engineering and Advanced Computing

Abstract. Hybrid memory, which leverages the benefits of traditional DRAM and emerging memory technologies, is a promising alternative for future main memory design. However popular management policies through memory-access recording and page migration may invoke non-trivial overhead in execution time and hardware space. Nowadays, managed language applications are increasingly dominant in every kind of platform. Managed runtimes provide services for automatic memory management. So it is important to adapt them for the underlying hybrid memory.

This paper explores two opportunities, heap partition placement and object promotion, inside managed runtimes for allocating hot data in a fast memory space (fast-space) without any access recording or data migration overhead. For heap partition placement, we quantitatively analyze LLC miss density and performance effect for each partition. Results show that LLC misses especially store misses mostly hit nursery partitions. Placing nursery in fast-space, which is 20% total memory footprint of tested benchmarks on average, causes only 10% performance difference from all memory footprint in fast-space. During object promotion, hot objects will be directly allocated to fast-space. We develop a tool to analyze the LLC miss density for each method of workloads, since we have found that the LLC misses are mostly triggered by a small percentage of the total set of methods. The objects visited by the top-ranked methods are recognized as hot. Results show that hot objects do have higher access density, more than 3 times of random distribution for SPECjbb and pmd, and placing them in fast-space further reduces their execution time by 6% and 13% respectively.

Keywords: Hybrid memory, Managed runtime, JVM, Memory management

1 Introduction

As processor cores, concurrent threads and data intensive workloads increase, memory systems must support the growth of simultaneous working sets. However, feature size and power scaling of DRAM is starting to hit a fundamental

limit. Different memory technologies with better scaling, such as non-volatile memory (NVM), 3D-stacked and scratchpad memory, are emerging. To leverage the benefits of different technologies, with disparate access-cost modules into an integrated hybrid memory opens up a promising future for memory design. It has the potential to reduce power consumption and improve performance at the same time [1]. However, it exposes the complexity of distributing data to appropriate modules.

Many hybrid memory management policies are implemented in a memory controller with or without OS assistance [2–9]. However, their page migrations can cause time and memory bandwidth overhead. There is also hardware space cost for recording memory access information which limits the size of management granularity too.

For portability, productivity, and simplicity, managed languages such as Java are increasingly dominant in mobiles, desktops, and big servers. For example, popular big data platforms, such as Hadoop and Spark, are all written in managed languages. Managed runtime provides services for performance optimization and automatic memory management. So it is important to adapt managed runtime for hybrid memory system.

This paper explores two opportunities: heap partition placement and object promotion, inside managed runtime for efficient hybrid memory management without additional data migration overhead. Hot objects (objects with high LLC miss density, i.e. LLC misses / object size) identification is conducted offline, thus no online profiling cost. We steal one bit from an object header as a flag to indicate hot object, so no impact on total space usage even at object grain. Our work is orthogonal to the management policies proposed inside an OS or hardware. They can work cooperatively but with reduced cost for managed applications. It can also work alone as a pure software portable hybrid-memory management.

For appropriate heap partition placement, we quantitatively analyze the LLC miss density for each partition of generational GC, including nursery, mature, metadata, and LOS (large object space). We demonstrate that the heap partitions according to object lifetime and characteristics also provide a natural partially classification of hot/cold objects. A 16 MB nursery covers 76% of total LLC misses, and most of them are store misses. Placing nursery in fast-space use 20% total memory footprint on average as fast-space, but only 10% performance difference from all heap in fast-space.

Besides the nursery, for workloads with relatively high LLC misses in mature partition, a small amount of the mature partition is allocated in fast-space to place hot objects during object promotion (which moves long-lived objects from nursery to mature partition). We develop an offline tool using the ASM bytecode manipulation framework [10] to record LLC miss density for each method. This information is used to direct JIT-generated machine-code to mark objects dereferenced by top-ranked methods as hot, so that they can later be moved to fast-space during object promotion. Results show that hot objects do have higher LLC miss density, more than 3 times of random distribution for SPECjbb and pmd, and placing them in fast-space further reduces their execution time by

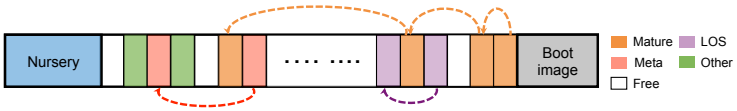


Fig. 1: Virtual memory space partition of Jikes RVM

6% and 13% respectively, resulting in ultimately 27% and 31% faster using our policy compared to the default OS policy of interleaving page allocation.

The structure of the paper is as follows. Section 2 and 3 are background for managed runtimes and related work. Section 4 is the management scheme we proposed. Section 5 introduces our hybrid memory emulation, and experimental settings. Finally, we discuss and conclude the results.

2 Managed runtime background

Managed runtime Managed-language applications require support services in order to run properly, such as Just-In-Time compilation (JIT) and Garbage collection (GC). Bytecode or only partially optimized code that are frequently executed are translated by the JIT into more optimized code. GC is used to automatically manage memory, including object allocation, and the identification and collection of unreachable objects. Among all types of GC, generational GC is the most popular one as it tends to reduce the overall burden of GC. It does this by partitioning the heap according to object lifetimes.

Generational GC and heap partition Since most objects have very short lifetimes, generational GC uses a low overhead space (called *nursery* space) for initial allocation and usage, and only moves objects that survive more frequent early collections to a longer lived space (called *mature* space). When a nursery is full, a minor GC is invoked to collect this space. When a minor GC fails due to a lack of space in mature space, a major GC will be performed. Nursery space is normally much smaller than mature space for efficient collection of short-lived objects. Jikes RVM also uses generational GC for its most efficient production configuration. This paper will use this configuration too. Under the production setting for the Jikes RVM, the heap uses a bump-allocation nursery and an Immix [11] mature space.

Other than nursery and mature spaces, Jikes RVM has spaces for large objects (LOS) including stacks, metadata (used by GC), as well as some small spaces for immortal (permanent), non-moving, and code objects, all of which share a discontiguous heap range with the mature space through a memory chunk free list. Figure 1 shows the virtual memory space partition of Jikes RVM. This paper will show how to distribute those partitions to appropriate memory space.

3 Related work

Different hybrid/heterogeneous main memory structures have been proposed, such as DRAM + NVM [12, 13, 6, 14, 3, 15, 4], on-chip memory + off-chip memory [16], and DRAMs with different feature values [17, 8]. Though with different

technologies, all of them share the common characteristic that different parts of the address space yield different access cost. Many technologies have been proposed to manage this heterogeneity.

Page access monitoring and migration Most of works use memory controllers to monitor page accesses [16, 6, 3, 4]. The memory controller will then migrate top-ranked pages to fast-space, and OS will update the virtual address mapping table accordingly. However, these research all operate at a fixed grain, mostly on a page level. Besides, page migration invokes time and bandwidth overhead. Bock et al. [12] qualify that page migration can increase execution time by 25% on average. Our work utilize object promotion for object placement to suitable space, thus does not invoke extra data migration nor time overhead.

Direct data placement To avoid data migration overhead, there are works that place data in appropriate space directly according to their reference behaviours [17, 14, 8, 15]. Chatterjee et al. [17] organize a single cacheline across multiple memory channels. Critical word (normally the first word) in a cacheline will be placed in a low-latency channel. Wei et al. [15] show that for the groups of objects allocated at the same place in the source code, some of them exhibit similar behaviours, which can be used for static data placement. Li et al. [14] develop a binary instrumentation tool to statistically report memory access patterns in stack, heap, and global data. Liu et al. [18] also implement a tool in OS to collect page access frequency, memory footprint, page re-use time, etc. which can be used to identify hot pages. Phadke and Narayanasamy [8] also profile applications' MLP and LLC misses offline to determine from which memory space each application would benefit the most. There are also works require programmers to help decide data placement [13, 19, 20]. Our paper uses offline profiling for data placement instead of online monitoring and migration, and does not invoke extra burden to programmers.

Adaption of managed runtime There are a few works related to the management of hybrid memory particularly for managed applications [21–26]. Shuichi Oikawa's group conduct some preliminary work [21–24]. They state that nursery should be in the fast memory space, but without supporting data, nor for metadata, stacks or other partitions in the heap. They extend write barriers to record the writes to each object online to find hot objects. However that will include write hit to cache which can mislead the data placement. Besides, the process could result in a non-trivial execution overhead. Inoue et al. [25] identify a code pattern in Java applications that can easily causing L1 and L2 cache misses. However, we find the number of memory accesses caused by this pattern is negligible. A series of papers [27–29] describing managed runtime techniques focusing on avoiding page faulting to disk. Those techniques could be applied to vertical hybrid main memory, where fast-space (e.g. DRAM) acts as a cache for slow-space (e.g. NVM). However Dong et al. [16] show that if performance difference between fast-space and slow-space (typically seen in DRAM and NVM) is not obvious, it may not be adequate for vertical hybrid memory to be viable. Hertz et al. [28] keep frequently used partitions such as nursery in memory, preventing them from being swapped, we show that this is still very significant for

the nursery even in our context. This paper profiles LLC miss density for each partition and method offline to decide object placement during a GC process.

4 Hybrid memory management scheme

4.1 Overview

The goal of our management system is to enable a machine with a small proportion of fast-space to perform as well as a machine with the same amount of memory where all the memory is fast-space. Specifically, we want hot objects be allocated in fast-space. We exploit two opportunities in managed runtimes: partition placement and object promotion. Analysing the LLC miss density of the various partitions, we determine if all, none or part of a partition is placed in fast-space. Object promotion to the mature partition provides a decision point where an object must be moved, as such, a cheap opportunity for moving to either slow-space or fast-space. Selecting either fast-space or slow-space is driven by offline profiling to determine hot methods (highest LLC miss density) and online marking of objects accessed by those methods. Compared to work before, our scheme does not cause online monitoring or extra data migration, thus no relevant performance overhead.

Heap partition placement To profile each heap partition, we use numactl library to map virtual memory space to fast-space or slow-space, and use PMU counters to measure LLC load/store misses. As explained in Section 2, the nursery is a continuous space, and we map it as a whole. LOS, metadata, and mature partitions share a dis-contiguous memory range divided into chunks. Free chunks are maintained on a list and available for any partition to allocate. To support hybrid memory, we change to two freelists, one for slow-space and one for fast-space. Different partitions will be given the right to access a specific list or both lists. When no free chunks are available in the fast-space list, objects will be allocated from the slow-space list. The overhead of managing the additional list is negligible.

In Section 6, we will show that the LLC miss density of the nursery partition is much higher than other partitions, and as such it is mapped to fast-space. A small proportion of the mature partition is mapped to fast-space, and hot objects are moved to that portion when they are promoted.

Object promotion Minor GC promotes surviving objects from the nursery to the mature partition. During this process, our scheme promotes hot objects to fast-space, and all other surviving objects are promoted to slow-space. Section 3 discusses some related work for finding hot objects. However, our experiments show that they either have a large overhead in time and space, are not effective, or need a programmer to annotate the critical data. Dynamic optimization prevents popular instrumentation tools, such as VTune, from being effective for profiling hardware events.

We develop a tool called HMprof to profile LLC miss density in the mature partition for each method of an application. Since our experiments show that even though each application executes hundreds or thousands of methods, only a

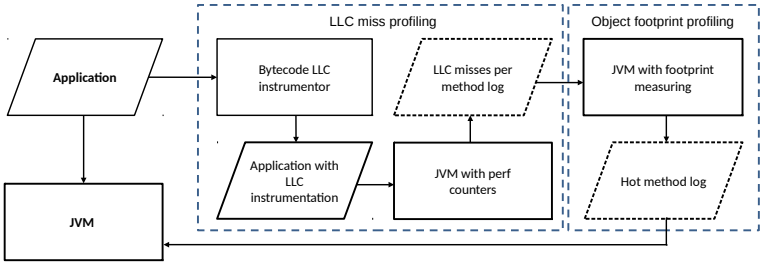


Fig. 2: Offline LLC miss per method profiling feeds into method footprint profiling which is then used to facilitate object promotion during normal execution.

dozen of them are responsible for most of the LLC misses, for example 0.4% and 1% of the methods in pmd and hsqldb covers 51% and 54.8% LLC respectively. This is a result of the particular access patterns and role of those methods. We will describe the implementation details next.

4.2 HMprof offline performance instrumentation

This instrumentation and profiling take place in two phases: first, collecting each method's performance counter statistics of any heap partition by instrumenting the application's java bytecode; and second, collecting the object visitation statistics for the candidate methods by altering the JIT. Figure 2, shows two offline profiling stages, and subsequence execution(s) of the application. These phases are described below.

LLC miss profiling Performance counter instrumentation is inserted into each method. Since methods may be compiled, and recompiled at runtime it is not possible to use the code location to identify the method, as such each method instrumented is given a unique numeric identifier as part of its instrumentation. The method code is transformed so that it has an additional local variable which is used to record a performance counter entry-value or re-entry value for the method, by invoking `perf.readPerf(I)J`, see Figure 3a. Upon exiting the method (either by return from the method, throwing an exception, or invoking a method), the entry-value, performance counter exit-value and the method identifier are logged.

The performance counters read and the values logged are maintained per thread. Logging a measurement is done by invoking `perf.updatePerf(IIJ)V`, see Figure 3b, which adds the performance counter delta for that thread to a thread specific array. Each element of the array corresponds to an instrumented thread identifier. Since there is no interaction between the threads, there is little impact overall, aside from a small perturbation of the cached data.

The statistics accumulated by a particular thread are finally added to the totals when that thread terminates. This imposes some cost on thread termination, but this does not have a significant performance impact. When all threads

```
/** read counter 0, save
to local variable e.g. 3 **/

iconst_0
invokestatic perf.readPerf(I)J
lstore_3
```

(a) Execution entry point

```
/** Update local variable
3 for method 576 **/
iconst_0
sipush  576
lload_3
invokestatic perf.updatePerf(IIJ)V
```

(b) Execution exit point

Fig. 3: Example instrumentation in a method.

```
getfield    Result(RefType) = ObjRef,Offset,field
byte_load   t0(B) = ObjRef,HOT_BIT_OFFSET
int_or      t1(B) = t0(B),HOT_BIT_MASK
byte_store  t1(B), ObjRef,HOT_BIT_OFFSET
```

(a) getfield

```
null_check  Result(GUARD) = Result(RefType)
byte_load   t0(B) = Result(refType),HOT_BIT_OFFSET
int_or      t1(B) = t0(B),HOT_BIT_MASK
byte_store  t1(B),Result(refType),HOT_BIT_OFFSET
```

(b) null_check

Fig. 4: Example HIR after inserting hot object marking instructions.

have terminated the final statistics values are written to a file. We can get a ranked list of methods according to their LLC misses in the mature partition.

Object footprint profiling Only LLC miss information is not enough to decide the hotness of a method, because the high LLC misses may be because it accesses a large amount of data. So LLC miss density is needed. Object visitation instrumentation is used to track the number of objects visited and their corresponding footprint for the top-ranked methods. An extra word is added to the object header to track which ones of those methods have visited a particular object. When a method visits an object, one corresponding bit will be marked. Because those methods are only a few, one extra word is enough. An extra phase is added to JIT to add marking code, and those methods are forcibly compiled with JIT. Thus, all objects dereferenced by any method that is being tracked are marked.

At the end of execution, the mature partition is scanned for marked objects, this excludes objects that do not survive a nursery collection. The number and footprint of the objects dereferences are accumulated per method. With the LLC miss information from the first step, we can work out the hot method list. Note that all the work above by HMprof is done offline. It will not add overhead to application run.

4.3 Hot Object Marking

After hot methods are found, we will mark the objects they visit as hot and move them to fast-space while an application runs. Similar as Section 4.2, we add a new phase in the HIR optimizations of JIT to mark the objects. This one flag bit is stolen from each object header. We target `getfield`, `putfield`,

and `call` as three of the most common cases where objects are dereferenced / visited. Importantly a reference that is returned from a call or a getfield has the potential to be dereferenced, if such dereference is to occur then before the return value is dereferenced, the HIR inserts a `null_check`. When a `null_check` of this form is seen then that too is instrumented. In general, these will cover most of the operations that cause object and array dereferencing. Figure 4a shows the additional instructions that are inserted after the `getfield`, note, `ObjRef` reference will already have passed a `null_check` or equivalent to get to this point, so it is safe to use. A similar pattern is used for `putfiled` and `call` (for virtual and interface calls). Similarly, we show in Figure 4b the additional code added after a `null_check`. In both cases they will set (mark) the hot bit in the object header as `true`, and in later versions will only conditionally set the hot bit.

We used replay compile in our experiment, so the JIT cost will not be counted when the application runs. However, as Cao et al.[30] shows that the JIT can run asynchronously with application threads, the application running time won't be sensitive to the JIT performance.

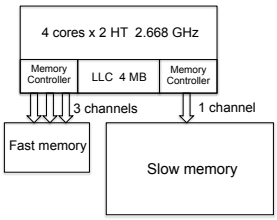
5 Experimental methodology

Hybrid memory emulator Because we do not have hybrid memory products and software simulators are very slow to run complex managed applications, we use a two-socket NUMA server (Intel Xeon X5650) with an interference program to mimic a hybrid memory structure. The interference program only runs on the remote CPUs, which regularly reads data from remote memory to increase the remote access latency and reduce its bandwidth for other applications. Tested benchmarks only run on the local ones.

Running the interference program and also using three channels for the local memory and a single channel for the remote memory, the latency of the mimic slow-space (remote memory) is 3.4 times that of the fast-space (local memory). The fast-space's read and write bandwidth is 5.3 times and 18.2 times those of the slow-space respectively, which is in keeping with current NVM characteristics—the write bandwidth is much worse than read. Those numbers fall within the range of latency and bandwidth characteristics for current NVM technologies [9, 2].

LLC size control The paper is about memory management policy design, and the memory footprint of the benchmarks is not large. To avoid the interference of LLC with the policy evaluation, we need to reduce the LLC size as much as possible. The method we used is mentioned in [31] as cache polluting method to reduce the LLC cache size to 4 MB. Based on the settings above, Figure 5a shows our emulated experimental platform.

Virtual machine configurations We implemented our approach in the Jikes RVM. The mature space is set so that it is big enough to hold all objects that survive nursery GC eliminating the effects of major GC. We use replay compilation to remove the nondeterminism of the adaptive optimization system. It also means that in our experiments, we will not include any JIT processing time.



(a) Platform

Workload	Description	Working set (MB)	Workload	Description	Working set (MB)
antlr	Parser and translator generator	44.1	avrora	Simulates the AVR microcontroller	43.3
bloat	Java bytecode optimization and analysis tool	63.6	eclipse	Integrated development environment	499.1
fop	Output-independent print formatter	43.3	hsqldb	Transaction processing	112.7
jython	interprets a the pybench Python benchmark	92.4	luindex	A text indexing tool	40.1
lusearch	Text search tool	397.7	pmd	Source code analyzer for Java	119.7
sunflow	Photo-realistic rendering system	62.7	xalan	transforms XML documents into HTML	337.6
SPECjbb2005	Transaction processing	326.8			

(b) Benchmarks

Fig. 5: Experimental platform and benchmarks.

We use DaCapo and SPECjbb benchmarks [32] for our experiments. Figure 5b describes each benchmark and their working set size.

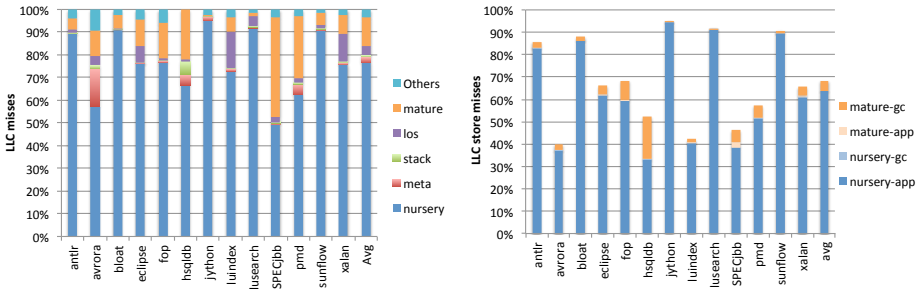
Default OS memory policy The default Linux memory policy interleaves allocation across all nodes in a NUMA system [33]. This policy will compare with ours in the performance evaluation. In our experimental platform, we use *mbind()* system call to set a VMA (Virtual Memory Area) policy, and interleave page allocation in a task's virtual address to fast and slow memory modules.

6 Evaluation results

6.1 Heap partition placement

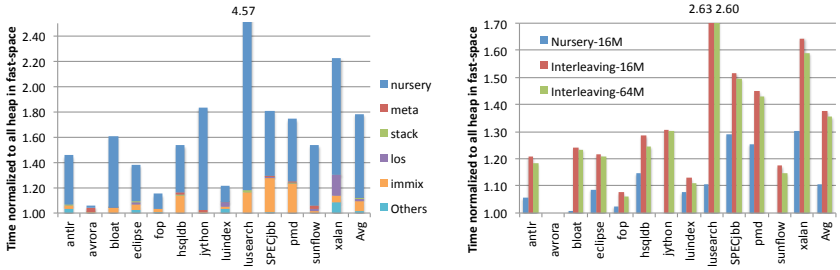
To decide how to place each partition in hybrid memory, we quantitatively evaluate the LLC misses associated with each partition, as well as the performance effect when it is placed in fast-space. Figure 6a shows the LLC miss distribution for each partition. It shows that though nursery is only 16 M, it covers the most LLC misses, 76% on average. This is primarily a result for all objects, except for large objects, being initially allocated in the nursery and causing a significant number of LLC misses during object initialization. We evaluate various nursery sizes from 4 M to 128 M. Results show that LLC misses due to nursery accesses increase with the nursery size. However, the curve flattens after 16 M for most applications, so we pick 16 M in the experiments. The mature partition covers 13% of the LLC misses on average. This percentage is relatively high for SPECjbb, pmd, and hsqldb, 44%, 27% and 22% respectively. The LLC misses hitting other partitions are small. Even though stacks are frequently accessed, they are mostly in cache, so will not cause many LLC misses. With the memory footprint result of total objects allocated in each partition, we compute the LLC miss density of the nursery to be 12.5 times that of the mature partition, 25 times that of LOS, and 33.3 times that of the metadata partition.

Some memory technology like NVM has a large write cost. Our hybrid memory emulator also sets the write bandwidth much smaller than that of the read bandwidth. Figure 6b shows the portion of store misses in total hitting the nursery and the mature partition when an application or GC executes. We can see that: a) for Java benchmarks, most LLC misses are store misses, 64% on average;



(a) (b)

Fig. 6: (a) LLC misses distribution; (b) LLC store misses %.

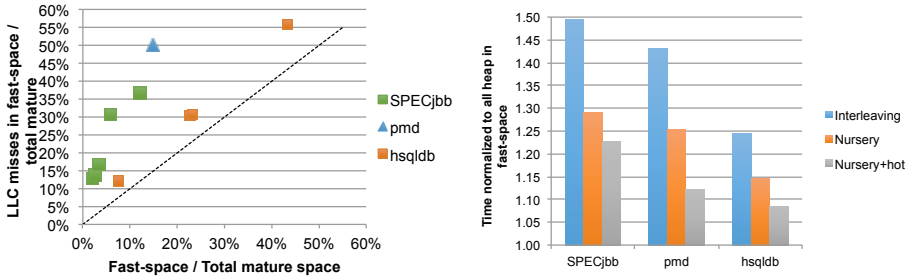


(a) (b)

Fig. 7: (a) Time effect of each partition, (b) Time comparison between nursery in fast-space and OS interleaving allocation.

b) almost all of these store misses hit in the nursery; c) in the nursery, nearly all store misses happen when application code executes, while in the mature partition, they happen during GC execution. Since the mature partition is set to be large enough that no major GC is triggered, all these store misses are a result of object promotion during minor GCs. So store misses are distributed evenly in the mature partition, which has been confirmed in our experiment results. This is undesirable for hybrid memory management especially for NVM, which expect the existence of write-hot data to put into fast-space.

Figure 7a is a stacked column chart showing the time effect of placing each partition in fast-space. The reference is the time when all heap is placed in fast-space. The total height of a bar shows the relative time when all heap is placed in slow-space. It reflects the LLC miss rate (LLC misses per instruction) of an application. lusearch has the highest LLC miss rate, 7.87 per 1K instructions. The second is xalan, 2.57 per 1K instructions. So when the heap is placed in slow-space, compared to being placed in fast-space, the running time is 4.57 and 2.22 times respectively. avrora has the lowest LLC miss rate, 1.03 per 1K instructions. So the time is basically the same when the heap is either in fast-space or slow-



(a) (b)

Fig. 8: (a) The correlation of LLC misses hitting in fast-space and the fast-space used, (b) time comparison of interleaving, nursery in fast-space, and nursery plus hot objects of mature partition in fast-space.

space. The figure also shows that corresponding to Figure 6a, placing a 16 M nursery into fast-space can greatly reduce the time, on average from 1.76 to 1.10, only a 10% difference from placing all the heap in fast-space. Placing the mature partition in fast-space can further reduce time from 1.08 to almost the same as all in fast-space (1.00) on average. For SPECjbb and pmd, the mature partition in fast-space reduces their normalized running time from 1.29 to 1.01, and 1.23 to 1.00.

We then compare the execution time of the nursery in fast-space with the default page interleaving scheme of OS in Figure 7b. For the interleaving scheme, we compare both a 16 M and larger 64 M interleaved fast-space setting with our 16 M setting. However, in both instances as interleaving treats every partition evenly, the time is much more than only the 16 M nursery in fast-space, 1.38 and 1.35 respectively compared to 1.10 on average.

All the results above suggest that the nursery should be placed directly into fast-space. It will get rid of the online monitoring and page migration costs. For mature space, we will allocate a small amount of fast-space for the hot objects found by our tool HMprof. The results will be analyzed in the next section.

6.2 Hot object allocation

Marked hot objects that are promoted to the mature partition are placed in fast-space, if available. In this subsection we show the effectiveness of our scheme, i.e. the objects moved to fast-space have high memory accesses but relatively small space cost. In particular SPECjbb, pmd and hsqldb are examined since they have more LLC misses in the mature partition than other benchmarks.

Figure 8a shows the correlation of fast-space usage and the LLC misses it covers when hot objects in mature partition are allocated in fast-space. pmd benefits most, with the hottest method only using 15% (12 M) of the mature partition objects (by volume) while covering 50% of the LLC misses. The hottest 12 methods of SPECjbb use 6% (18 M) of the mature partition objects (by volume) and cover 31% of the LLC misses. After the first 12 methods the thirteenth is proportionally less effective, resulting in a total use of 12% (36 M) of the mature

partition objects (by volume) and 37% of the LLC misses being covered. The remaining methods are not considered hot or account for too few LLC misses. The hot methods for hsqldb are not obvious, and with the 12 hottest methods using 43% (42 M) of the mature partition objects (by volume) while covering 56% of the LLC misses, only a little better than random promotion. As noted in Section 6.1, the distribution of write LLC misses amongst objects in the mature partition is more even than that of read LLC misses, and given the cost difference between read and write, it is adversely affecting our performance improvements.

Figure 8b compares relative execution time of default OS page interleaving policy using 64 MB fast-space, only a 16 M nursery in fast-space, and both the 16 M nursery and hot objects of mature partition in fast-space. The policy proposed gets the best performance for the three benchmarks. For SPECjbb and pmd only 16% and 23% (respectively) of their working sets reside in fast-space (including the nursery partition), while only degrading performance by 23% and 12% respectively from that of all the heap in fast-space. Placing 51% of the working set for hsqldb in fast-space yields a performance degradation of 8% over than of all the heap in fast-space.

7 Conclusion

Hybrid memory is a promising alternative for future memory systems, however, its complexity challenges effective management. This paper explores unique opportunities inside managed runtimes for efficient and portable hybrid memory management with negligible online overhead. Our policy places the nursery and a small amount of the mature partition in fast-space. Compared to the default OS policy, for SPECjbb and pmd, our scheme is 27% and 31% faster. Compared to placing all the heap in fast-space, we place 16% and 23% heap in fast-space, but achieve only 23% and 12% performance difference respectively for the two benchmarks. In future work, we will target big data applications as test benchmarks.

Acknowledgement This work is supported by the National High Technology Research and Development Program of China (2015AA011505), the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing (2016A03), the China Postdoctoral Science Foundation (2015T80139), the National Natural Science Foundation of China (61202055, 61221062, 61303053, 61432016, 61402445, 61432018, 61133005, 61272136, 61521092, 61502450).

References

1. Xie, Y.: Modeling, architecture, and applications for emerging memory technologies. Design Test of Computers, IEEE **28**(1) (Jan 2011) 44–51
2. Qureshi, M.K., Srinivasan, V., et al: Scalable high performance main memory system using phase-change memory technology. In: ISCA '09, ACM (2009) 24–33
3. Ramos, L.E., Gorbatov, E., et al: Page placement in hybrid memory systems. In: ICS, ACM (2011) 85–95

4. Zhang, W., Li, T.: Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In: PACT '09. (2009)
5. Li, Y., Choi, J., et al.: Managing hybrid main memories with a page-utility driven performance model. In: arXiv.org. (Jul. 2015)
6. Dhiman, G., Ayoub, R., Rosing, T.: PDRAM: A hybrid PRAM and DRAM main memory system. In: DAC '09. (July 2009) 664–669
7. Yoon, H., Meza, J., Ausavarungnirun, R., Harding, R., Mutlu, O.: Row buffer locality aware caching policies for hybrid memories. In: ICCD. (2012) 337–344
8. Phadke, S., Narayanasamy, S.: MLP aware heterogeneous memory system. In: DATE, IEEE (2011) 956–961
9. Caulfield, A.M., Coburn, J., et al.: Understanding the impact of emerging non-volatile memories on high-performance, io-intensive computing. In: SC. (2010)
10. OW2 Consortium: ASM (2015)
11. Blackburn, S.M., McKinley, K.S.: Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In: PLDI. (2008)
12. Bock, S., Childers, B.R., Melhem, R.G., Mossé, D.: Concurrent page migration for mobile systems with os-managed hybrid memory. In: CF, ACM (2014) 31:1–31:10
13. Dulloor, S., Roy, A., et al.: Data tiering in heterogeneous memory systems. In: EuroSys, ACM (2016) 15:1–15:16
14. Li, D., Vetter, J.S., et al.: Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. In: IPDPS, IEEE (2012) 945–956
15. Wei, W., Jiang, D., McKee, S.A., Xiong, J., Chen, M.: Exploiting program semantics to place data in hybrid memory. In: PACT, IEEE (2015) 163–173
16. Dong, X., Xie, Y., et al.: Simple but effective heterogeneous main memory with on-chip memory controller support. In: SC. (2010) 1–11
17. Chatterjee, N., Shevgoor, M., et al.: Leveraging heterogeneity in DRAM main memories to accelerate critical word access. In: MICRO, IEEE (2012) 13–24
18. Liu, L., Li, Y., Ding, C., Yang, H., Wu, C.: Rethinking memory management in modern operating system: Horizontal, vertical or random? IEEE Trans. Computers **65**(6) (2016) 1921–1935
19. Hassan, A., Vandierendonck, H., et al.: Software-managed energy-efficient hybrid DRAM/NVM main memory. In: CF, ACM (2015) 23:1–23:8
20. Malicevic, J., Dulloor, S., et al.: Exploiting NVM in large-scale graph analytics. In: INFLOW, ACM (2015) 2:1–2:9
21. Nakagawa, G., Oikawa, S.: Preliminary analysis of a write reduction method for non-volatile main memory on jikes rvm. In: CANDAR. (Dec 2013) 597–601
22. Nakagawa, G., Oikawa, S.: An analysis of the relationship between a write access reduction method for nvm/dram hybrid memory with programming language runtime support and execution policies of garbage collection. In: IIAIAAI. (2014)
23. Nakagawa, G., Oikawa, S.: Language runtime support for nvm/dram hybrid main memory. In: COOL Chips XVII, 2014 IEEE. (April 2014) 1–3
24. Nakagawa, G., Oikawa, S.: NVM/DRAM hybrid memory management with language runtime support via MRW queue. In: SNPD, IEEE (2015) 357–362
25. Inoue, H., Nakatani, T.: Identifying the sources of cache misses in java programs without relying on hardware counters. In: ISMM, ACM (2012) 133–142
26. Schneider, F.T., Payer, M., Gross, T.R.: Online optimizations driven by hardware performance monitoring. In: PLDI, ACM (2007) 373–382
27. Yang, T., Hertz, M., Berger, E.D., Kaplan, S.F., Moss, J.E.B.: Automatic heap sizing: taking real memory into account. In: ISMM, ACM (2004) 61–72

28. Hertz, M., Feng, Y., Berger, E.D.: Garbage collection without paging. In: PLDI, ACM (2005) 143–153
29. Yang, T., Berger, E.D., Kaplan, S.F., Moss, J.E.B.: CRAMM: virtual memory support for garbage-collected applications. In: OSDI, USENIX Association (2006) 103–116
30. Cao, T., Blackburn, S.M., et al.: The yin and yang of power and performance for asymmetric hardware and managed software. In: ISCA, IEEE (2012) 225–236
31. Ferdman, M., Adileh, A., et al.: Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In: ASPLOS, ACM (2012) 37–48
32. Blackburn, S.M., Garner, R., et al.: The dacapo benchmarks: java benchmarking development and analysis. In: OOPSLA, ACM (2006) 169–190
33. The Linux Kernel Organization: NUMA memory policy (2015)