Impact of Intrinsic Profiling Limitations on Effectiveness of Adaptive Optimizations[1]

MICHAEL R. JANTZ, University of Tennessee
FORREST J. ROBINSON, and PRASAD A. KULKARNI, University of Kansas

Many performance optimizations rely on or are enhanced by run-time profile information. However, both offline and online profiling techniques suffer from intrinsic and practical limitations that affect the quality of delivered profile data. The *quality* of profile data is its ability to accurately predict (relevant aspects of) future program behavior. While these limitations are known, their impact on the effectiveness of profile-guided optimizations, compared to the ideal performance, is not as well understood. We define *ideal* performance for adaptive optimizations as that achieved with a precise profile of *future* program behavior.

In this work we study and quantify the performance impact of fundamental profiling limitations by comparing the effectiveness of typical adaptive optimizations when using the best profiles generated by offline and online schemes against a *baseline* where the adaptive optimization is given access to profile information about the future execution of the program. We model and compare the behavior of three adaptive JVM optimizations – heap memory management using object usage profiles, code cache management using method usage profiles, and selective just-in-time compilation using method hotness profiles – for the Java DaCapo benchmarks. Our results provide insight into the advantages and drawbacks of current profiling strategies, and shed light on directions for future profiling research.

CCS Concepts: •**Software and its engineering** → **Runtime environments;** *Software performance;*

General Terms: Performance, Measurement, Languages

Additional Key Words and Phrases: profiling, profile-guided optimizations

ACM Reference Format:
Michael R. Jantz, Forrest J. Robinson, and Prasad A. Kulkarni, 2016. Impact of Intrinsic Profiling Limitations on Effectiveness of Adaptive Optimizations *ACM Trans. Embedd. Comput. Syst.* 9, 4, Article 39 (March 2016), 25 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Program *profiling* is a powerful technique to discover, understand and reason about the dynamic or run-time behavior of a program. Accurate, comprehensive, and timely profiling information can not only enhance the effectiveness of existing run-time algorithms, but also enable new adaptive or feedback-driven optimizations (FDO) resulting in significant improvements to program performance. Indeed, run-time systems, such as Java virtual machines (VM), have a critical need for profiling assisted algorithms, and employ them for many optimization tasks to benefit performance.

[1]**Extension of Conference Paper:** While not truly an extension, Section 5 of this paper borrows from our earlier conference paper [Robinson et al. 2016]. This current paper has an entirely different theme – including introduction, related work, conclusions, etc. – from our earlier conference submission. Apart from section 5, all other sections in this paper are original contributions of this work.

This work is supported by the National Science Foundation, under CAREER award CNS-0953268, and grants CCF-1619140, CCF-1617954, and CNS-1464288, and a 2016 Intel SSG award.
Author's addresses: Michael R. Jantz, Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville; Forrest J. Robinson, and Prasad A. Kulkarni, Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence.

DOI: 0000001.0000001

Profile information is typically collected using one of two strategies: (1) additional prior runs of the same program (*offline* strategy), or (2) dynamically, during the current program run (*online* strategy). Both online and offline profiling strategies suffer from several *intrinsic* and *practical* limitations. We define *intrinsic* limitations as those that exist even when the profile data obtained is as accurate and timely as is possible with that technique without any constraints on physical costs. We define *practical* limitations as those that are imposed by cost, overhead, and other physical factors. These limitations can affect the predictability of the obtained profile data as compared to the *ideal* profile that is generated with precise knowledge of the *future program behavior*.

Many static compilers, like GNU gcc/g++, provide the option of offline profiling based optimizations [Hwu et al. 1993; Mock et al. 2000; Chang et al. 1991; Pettis and Hansen 1990]. Although the resulting performance improvements are often significant, offline profiling based systems face the following intrinsic limitations: (a) a different input set or execution environment can cause the application's run-time behavior to differ from its behavior during the profiling run, and (b) profiling data collected during a separate run must be structured and aggregated so that it can be used by FDOs. Data aggregation can reduce the effectiveness of FDOs by limiting its ability to customize for different sections/phases of the program run. Additionally, offline profiling also suffers from the practical restriction presented by the difficulty or inability in certain cases to collect a profile trace of the application prior to execution. It is believed that an ability to perform the profiling at run-time using an *online* strategy may help overcome some of these drawbacks [Arnold et al. 2000a; Arnold et al. 2002].

Unfortunately, intrinsic limitations persist with online profiling. Adaptive optimizations desire accurate knowledge of *future* program behavior to be most effective. However, existing online profiling schemes are typically *reactive* (they can only monitor the program's past execution events). Thus, FDOs often use simple models that predict that a program's future execution will truly mirror its past behavior. Naturally, adaptive optimizations may perform quite poorly if this prediction turns out to be incorrect.

A practical problem with online profiling schemes is that the cost imposed by comprehensive profile collection at run-time may be prohibitively expensive, and could slow down overall program execution. In such cases, VMs use low-overhead techniques, such as *sampling*, and only gather partial behavior data that is quick and easy to collect at run-time. The result is often incomplete or inaccurate program profile information, which can reduce or even negate the benefit of adaptive feedback-driven optimizations. Additionally, collection of sufficient profile data after program start may delay the application of some FDOs, which can affect program performance.

These limitations with offline and online profiling are generally known. Yet, we do not entirely understand their impact on the effectiveness of FDOs. In this work we employ a set of common adaptive VM tasks/optimizations and conduct a *fundamental* study to assess, evaluate, and quantify the impact of the intrinsic profiling limitations. For each adaptive task, we develop a *baseline* where the task has access to profile information from the remaining *future execution of the program*. This methodological baseline is called *future profiling* in the rest of this paper. It is important to note that future profiling is not a new profiling strategy, but only serves as a baseline against which realizable offline and online profiling strategies can be evaluated. Additionally, future profiling is not an oracle, and adaptive tasks employing the future profile may still on occasions make sub-optimal optimization decisions.

We construct detailed experimental frameworks to conduct each study. These frameworks constrain the behavior of the VM to make our analysis more manageable. Each framework supplies the selected adaptive task with the best profile data that could be collected by offline, online, and future profiling techniques without cost/overhead considerations. The performance of each adaptive task with the different instances of

profile data are evaluated and compared. We also evaluate the performance achieved by each adaptive task for a standard online profiling technique that only has access to profile data that is limited by the physical and practical constraints of profile collection overhead and/or delay at run-time.

We employ three adaptive VM tasks for this work: (a) heap memory management that uses profiles of object access patterns to save energy, (b) code cache management that employs method usage profiles to reduce memory consumption, and (c) just-in-time (JIT) compilation that uses method hotness data to improve speed. These tasks were chosen as they are representative of adaptive optimizations performed by VMs.

The primary contribution of this work is a comparison of the potential benefits of basic profiling techniques and a deeper understanding of the performance impact of intrinsic profiling limitations for three adaptive VM tasks in the context of the DaCapo benchmarks and the HotSpot Java VM. We conduct the following tasks in this work.

— We design and build creative VM-based frameworks to compare the performance of adaptive VM tasks with the best profiles generated by offline and online strategies against the future profiling baseline. Our experimental frameworks allow us to effectively control profiling accuracy and cost, while achieving realistic comparisons.
— For offline profiling, we measure: (a) the performance impact of the difference in training and evaluation inputs on performance of FDOs, and (b) the effect of aggregating profile information across the entire program run.
— For online profiling, we: (a) assess the effectiveness of FDOs when they have access to the best online profiling data while ignoring any cost overhead, and (b) evaluate the impact of online techniques that only have access to incomplete profile information due to practical issues of run-time overhead and profile data collection delay.

2. RELATED WORK

In this section we survey past works in the areas of evaluating profiling accuracy, reducing online profiling overhead, exploring the predictability of program behavior, and the use of predictive models during VM tasks.

Researchers have attempted to evaluate the accuracy of (estimation-based) profilers. Anderson et al.'s experiments with their Digital continuous profiling framework revealed that sampling-based profilers must collect their samples randomly to achieve accuracy [Anderson et al. 1997]. Mytkowicz et al. compared the accuracy of four sampling-based Java profilers, and made a similar observation [Mytkowicz et al. 2010]. Researchers that develop a new profiling strategy also often attempt to evaluate its accuracy. In cases where a *correct* profile is available, the new profiling data can simply be compared with this correct profile [Arnold and Grove 2005; Duesterwald and Bala 2000; Moseley et al. 2007]. Arnold and Grove used this scheme for their profiling strategy measuring method call frequencies [Arnold and Grove 2005]. In other cases, a known correct profile may not be available because of the *observer effect*, where the act of collecting the profile affects the measured value. Timing data is an example of such a value. In such cases, researchers have used causality analysis to assess if their profile is *actionable*, i.e., acting on the profile yields the expected outcome [Mytkowicz et al. 2010; Rubin et al. 2002]. While establishing appropriate profiler accuracy is an implementation and cost issue, our quest in this work is different and more fundamental. We aim to understand and evaluate the limitations of offline and reactive online profiling schemes even *when the profiles that they generate are completely accurate*.

Generating comprehensive profile information at run-time can be intolerably expensive. Several works attempt to manage the overhead of online profiling, especially by using parallelism on multi-core machines. *SHIM* is a sampling-based fine-grain profiling tool that exploits unutilized hardware to execute a profiling thread with very

high profile resolution and low overhead [Yang et al. 2015]. Approaches like *shadow profiling* and *SuperPin* attempt to gather accurate profile data using instrumented slices that run in parallel with the original uninstrumented program on multi-core machines [Moseley et al. 2007; Wallace and Hazelwood 2007]. The PiPA approach performs low-overhead profiling in the same thread, but conducts the more heavy-weight profile analysis in separate threads [Zhao et al. 2008]. Whaley developed a low-overhead profiler that samples the program stack frame at arbitrary points using a different dedicated profiler thread [Whaley 2000]. Arnold and Ryder employ sampling-based bursty tracing to reduce overhead [Arnold and Ryder 2001]. The Java Virtual Machine Tool Interface (JVMTI) [Oracle 2014] supports both sampling and instrumentation for profiling some program events, is supported by most common Java VM implementations, but rarely provides a high-performance solution [Hofer and Mössenböck 2014; Binder 2006]. Other profiling approaches, including those using hardware performance counters [Inoue and Nakatani 2009], have also being developed earlier. Our goal in this work is different – we assess effectiveness of adaptive algorithms when they have access to the best online profiling data while ignoring any cost overhead.

Researchers have studied the predictability of program behavior across different inputs and its variability during the same run. Berube studied issues relating to input-dependent program behavior to enhance FDO performance in ahead-of-time compilers [Berube 2012]. This study is particularly relevant to our work as it addresses the challenge of workload selection and profile data aggregation from multiple training runs during offline profiling. Other researchers have found that inputs may need to be specifically designed to ensure reasonably representative profiles, and even then the results are not always as desired or ideal. For example, high variability was reported between the specially designed *training* and *reference* input sets for several benchmarks in the popular SPEC cpu2000 [Hsu et al. 2002] benchmark suite, and for a few of the SPEC cpu2006 [Gove and Spracklen 2007] benchmarks. Another study observed that programs typically exhibit significant variability in behavior even at the level of millions of instructions during the same program execution [Duesterwald et al. 2003], concluding that reactive online profiling approaches may be inadequate. These studies reveal the challenges facing FDO, especially those relying on offline profiling that not only need to use profiles from distinct program runs/inputs, but also aggregate the profile information over the entire run before its application to FDO.

Many FDOs employ simple prediction models that assume future behavior to faithfully reflect the past [Arnold et al. 2000b]. Earlier research suggests that such *reactive* online profiling techniques may not be able to accurately predict future behavior [Duesterwald et al. 2003; Namjoshi and Kulkarni 2010]. Some works derive enhanced knowledge of future behavior in targeted situations. For instance, knowledge of loop bounds was used to predict function hotness [Namjoshi and Kulkarni 2010]. Machine learning was used to correlate certain program behaviors with values of input parameters to guide early and better optimization decisions [Jiang et al. 2010]. This learning scheme was later extended to correlate the behaviors of unrelated loop bounds to predict program behavior patterns and improve dynamic method version selection [Wu et al. 2012]. Static analysis has also been used to reduce overhead of online profiling based GC optimizations [Huang et al. 2004]. A priori knowledge of method compilation and execution times was used to derive an ideal method compilation ordering to improve VM performance [Ding et al. 2014]. Researchers have also attempted to find and exploit the periodicity in program behavior (program *phases*), for instance, to enhance online predictor accuracy [Duesterwald et al. 2003]. While we don't evaluate the effectiveness of such predictive schemes, our work provides further motivation for these and other related techniques to improve the ability of profiling schemes to predict future program behavior more quickly, accurately and at lower costs.

3. COMMON EXPERIMENTAL SETUP

We design innovative and detailed VM-based frameworks to conduct our experiments in this work. These frameworks provide a controlled environment to study profiling properties and understand their impact on FDOs. We construct our profiling models in this framework using traces from actual VM runs.

Our experiments use Oracle's production-grade HotSpot Java virtual machine to collect trace information [Paleczny et al. 2001; Kotzmann et al. 2008]. We extend HotSpot (JDK-6 or JDK-9 depending on available partial frameworks from our earlier works) to collect the required profile data for each selected task, as well as to run any validation experiments. In this section we provide a brief background on the internal workings of HotSpot that are relevant to our current work.

HotSpot includes a high-performance threaded bytecode interpreter and multiple JIT compilers. The execution of a new program begins in the interpreter. HotSpot uses each method's *hotness_count*, which is a sum of the method's *invocation* and loop *backedge* counters, to promote methods to (higher levels of) JIT compilation. We insert our own invocation and backedge counters in the HotSpot interpreter that are incremented at all times outside of the profiling flags. The native code generated after compilation is stored in a region of heap memory called the *code cache*. HotSpot and our constructed frameworks do not currently detect or employ program phases.

Data objects (class instances) created by the Java program at run-time also occupy a region of heap memory. Periodically, the heap area will fill with objects created by the application, triggering a garbage collection (GC). During GC, the VM frees up space associated with the dead (unreachable) objects, and possibly shrinks or expands the virtual memory space depending on the current need.

Our experiments employ the DaCapo Java benchmarks [Blackburn et al. 2006], and we report results with their *default* input size. We also collect profiles of executions with the *small* input sizes, but this data is only used to make predictions with the offline-diff strategy (described in the following sections). The run-time experiments were performed on a cluster of Intel x86-64 machines running Fedora Linux as the operating system. All the run-time results in these sections report the (geometric) average over 10 runs for each benchmark-configuration pair [Georges et al. 2007].

4. HEAP MEMORY MANAGEMENT

Our first study examines the effectiveness of different profiling strategies to divide heap objects into sets according to their expected usage patterns. Such a capability may be employed during many optimizations. For example, researchers have used profiling of memory access patterns to improve locality by storing hot data closer together or in a cache-conscious way [Calder et al. 1998; Chilimbi and Shaham 2006; Huang et al. 2004; Sudan et al. 2010; Brock et al. 2013; Guo et al. 2015]. Partitioning objects/-data with different access patterns, capacities, and/or latency requirements is also an essential element of efficient execution on systems with heterogeneous memory architectures. Several recent projects have used profiling to guide memory management to improve performance, reduce power, and/or benefit durability in such systems [Chen et al. 2014; Chang et al. 2014; Meswani et al. 2015; Agarwal et al. 2015].

We explore the problem of employing profile information to organize heap objects into hot/cold sets to improve memory power management. Energy efficiency in memory is critically important in mobile and embedded domains, and in high-end domains, such as enterprise and scientific computing [Malladi et al. 2012; Lefurgy et al. 2003; Hoelzle and Barroso 2009; Lim et al. 2009]. According to one estimate [Lefurgy et al. 2003], memory accounts for about 40% of the energy consumed in a typical server – comparable to or slightly higher than the processors' contribution.

To reduce energy costs in memory, most modern systems include the ability to automatically transition individual memory devices into low-power states (such as "self-refresh") when they are not in active use [JEDEC 2009]. To amplify the effectiveness of this technique, it is often desirable to co-locate objects that will be accessed frequently (i.e. hot objects) onto their own small set of memory devices. This configuration allows the other memory devices, which are filled with cold objects that are not likely to be accessed, to transition to low-power states more often.

In previous work, we found significant potential to increase DRAM energy efficiency by coordinating object placement decisions across the hardware, lower-, and upper-level software [Jantz et al. 2015]. For this work, we repurpose our *cross-layer* framework to evaluate the effectiveness of different profiling strategies when applied to an optimization to reduce DRAM energy consumption. We presume the existence of a runtime (similar to the HotSpot-6 based framework from our earlier work) that is capable of periodically assigning heap objects to disjoint hot and cold spaces corresponding to different groups of memory devices. The optimization problem then is to maximize the size of the objects assigned to the cold space, while at the same time, ensuring that the number of accesses to such objects remains low enough to allow the corresponding "cold" memory devices to transition to low-power states. Thus, the effectiveness of this technique depends upon the ability to accurately predict future object access patterns.

4.1. Motivation

In this section we demonstrate the importance of *prediction accuracy* during heap object management for reducing energy consumption. For these experiments, we extend the MemBench benchmark from our earlier work [Jantz et al. 2015]. MemBench creates two types of objects: *HotObject* and *ColdObject*. Each object contains a 1MB array of integers, which represent the object's memory resources. The custom JVM has the ability to recognize the object type at allocation time, and allocate objects to different heap partitions based on their type.

On initialization, MemBench allocates a large number of hot and cold objects. It then spawns a software thread for each underlying hardware thread and divides the objects equally amongst them. The threads continuously iterate over the objects, selecting the next hot/cold object to access and writing data to each cell of its associated array.

In the ideal (perfect prediction) scenario, the accessor threads only write to objects of the HotObject type, allowing the devices in which the ColdObjects reside to frequently operate in low-power states. We further extend MemBench with an additional parameter that controls the likelihood that each accessor thread selects and accesses a ColdObject at each iteration. Controlling this parameter allows us to model the effect of mis-predicting the hotness/coldness of a set of application data objects.

Our experiments use a single socket of an HP ProLiant DL380p Gen8 server machine with an Intel E5-2620 v2 (Ivy Bridge) processor. This machine has 6 2.1GHz cores with hyperthreading enabled and 4 8GB DIMMs of HP DDR3 SDRAM (product #: 713979-B21), which are each connected to their own channel. We configure MemBench to allocate roughly 24 GB of cold program objects and 3.5 GB of hot program objects. The objects are allocated in random order. Hence, in the default configuration, HotObjects and ColdObjects are stored intermittently throughout the application heap, and ultimately, across all four memory modules. In the modified JVM, however, objects of the ColdObject type are allocated to a separate heap region, which is backed with physical memory corresponding to only 3 of the 4 DIMMs in the system. The remaining objects (including the HotObjects) reside on the other DIMM.

For each MemBench run, the threads iterate for 100 seconds, and use a delay factor of 200ns between memory accesses. Performance is recorded as the total number of (hot or cold) objects processed. We estimate energy consumption using the same

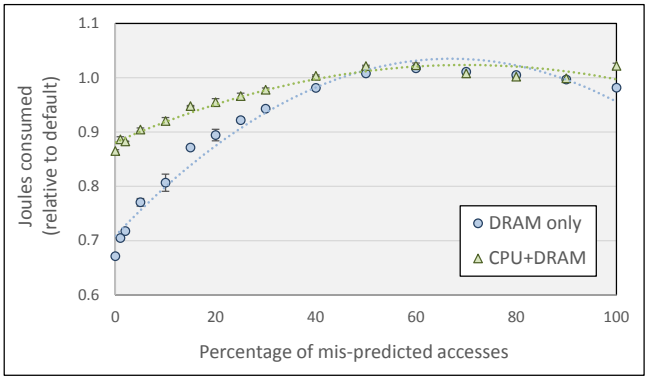

Fig. 1. Rising fraction of mis-predicted accesses can rapidly diminish/erase the benefit of the heap memory management optimization. (Lower is better is all the graphs in this paper.)

model [David et al. 2011] and tools employed in our previous work. Our results report the average performance and energy measurements over five experimental runs.

The design and configuration parameters of MemBench were chosen to model DRAM energy consumption with an application that allocates *most* of the system's memory, but only requires frequent access to a small portion (about 10% to 15%) of its objects. It is important to note, however, that the energy expense of memory does not directly depend on the size of the objects in memory, but instead, is mainly determined by the distribution of objects across the system's memory hardware and the rate and pattern of access to objects in memory. In a separate set of experiments, we configured MemBench to use smaller objects (24 bytes per object instead of 1MB) and less overall capacity (about 840MB in total), but did not change the distribution of objects across the DRAM devices or the rate or pattern of access to the objects. We found that this low capacity version of MemBench had very similar DRAM and CPU energy requirements as the high capacity version presented here.[2]

Figure 1 shows the DRAM and (CPU+DRAM) energy consumed while running MemBench on the adopted JVM framework, which assigns HotObjects and ColdObjects to separate memory devices, compared to a run with the default JVM. Each dot in each series presents the relative energy consumed with a different proportion of *mis-predicted* accesses (i.e., accesses to ColdObjects) along the X-axis.

With ideal prediction (no mis-predicted accesses), partitioning the hot and cold objects reduces DRAM energy consumption by 32.9%. However, increasing the percentage of mis-predicted accesses quickly diminishes the effectiveness of this optimization. For instance, we found that mis-predicting 20% of object accesses reduces the energy savings by about $\frac{2}{3}$ compared to the ideal prediction. Mis-prediction rates 40% or higher negate all of the efficiency improvements enabled by this technique.[3] These experiments reveal that *low-overhead and accurate profiling is critical to realizing potential energy savings with this optimization*.

[2]The high capacity version consumes 20.9 Watts of DRAM power (50.5W of CPU+DRAM power) with the default JVM and 14 Watts of DRAM power (43.6W of CPU+DRAM power) with the custom JVM with zero mis-predicted accesses. The low capacity version consumes 20.1 Watts and 13.5 Watts of DRAM power (50.5W and 44.0W of CPU+DRAM power) with the same default and custom JVM configurations, respectively.

[3]The observed energy reductions are solely due to a reduced rate of DRAM power consumption. Performance with the modified JVM is similar to or slightly worse than the default configuration, regardless of the mis-prediction rate. In the worst case, with 100% mis-prediction, performance is 6.5% worse than the default configuration. On average, performance degrades by 1.5% with the modified JVM framework.

4.2. Profiling for Hot and Cold Objects

In this section, we present experiments conducted using a framework we developed to compare the effectiveness of various profiling strategies to identify and co-locate objects with similar usage rates. Our framework first generates a trace of the total size and number of accesses to objects created at different static program allocation sites during each fixed-length interval of the program run. Our heap memory management algorithm can employ this trace data to partition the heap memory into hot/cold sets, which can then be assigned to different memory DIMMs. Different profiling strategies have access to and/or can employ different partitions of trace data before or during the program run to make optimization decisions. We study the impact these differences in the profiling strategies have on the effectiveness of our heap management algorithm as compared to one that has access to *future* trace information at each program point.

Evaluation Metrics:. Memory energy depends not only on the rate of usage, but also the number of devices in use. Thus, we consider both heap size and the number of accesses to objects in the cold set as evaluation metrics. Our model assumes that the system allows a *fixed* small fraction of memory accesses to the objects in the cold set to achieve energy efficiency. If the fraction of accesses to the cold set is greater than the maximum allowed by our configuration, then it suggests that the profiling strategy incorrectly assigned too many hot objects to the cold set, potentially preventing the "cold" memory devices from transitioning to low-power states. On the other hand, if the strategy turned out to be too conservative, and it assigns a large number of cold objects to the hot set, the system could miss potential opportunities for energy savings.

Methodology:. Our framework provides a profiling mechanism for recording the combined size and usage information of objects created at the same *program allocation site*. (see Section 7.1 of [Jantz et al. 2015]). Using this mechanism, we record the memory usage activity of all but two benchmarks in the DaCapo suite.[4] Each profile run prints the heap usage information at regular, timer-based intervals. We experimented with a range of timer intervals and found that 250msec provides a good balance between the compute/memory resources required to perform the profiling and experiments in our framework while also providing an adequate number of samples. Results with the different intervals are presented in Figure 6. All other results in this section were collected using a 250msec interval.

Table I presents information on the DaCapo benchmarks used in our studies. Columns 3 – 7 show heap usage statistics that were collected with our profiling framework, specifically: the number of profiling intervals, the number of allocation sites reached during the run, the total size of objects allocated on the heap, the maximum heap size recorded at one interval, and total heap accesses. Hence, the selected benchmarks exhibit a wide range of capacity and usage requirements.

We express the problem of partitioning the profiled allocation sites into hot and cold sets as an instance of the 0/1 knapsack optimization problem. We implement a number of different partitioning strategies by varying the profile information provided as its input. To compare partitionings with different input profiles, we select knapsack capacities as a percentage of the total number of accesses in the input profile. For example, a knapsack with a capacity of 5% selects allocation sites that account for no more than 5% of the total number of accesses in the input profile. We select five different knapsack capacities, (1%, 2%, 5%, 10%, and 20%), and evaluate each partitioning strategy with each capacity.

[4]We omit *tradebeans* and *tradesoap* from our study because profiling these benchmarks with the *default* input fails before completing one full iteration. In both cases, the failure is due to a timeout exception that occurs while the application is reading a socket.

Table I. Benchmark statistics. The columns from left to right show the: benchmark name, run-time in seconds, # of heap profile intervals (with an interval length of 250msec), # of allocation sites reached during the run, total size of heap allocations (in KB), maximum heap size (in KB), total # of heap accesses (R/W), # of invoked methods, # of hot methods, and the size occupied by the hot compiled code (in KB). †Run-time is measured using an unmodified HotSpot-6 with default parameters. ‡Heap usage is collected using the framework (with HotSpot-6 as base) from Section 4.2. §The number of invoked and hot methods, and size of compiled code are recorded using HotSpot-6 with the default c1 compiler.

Benchmark	RT (s)†	Heap Usage‡					Methods§		
		#PIs	#Sites	Alloc. KB	Max KB	Acc.'s (R/W)	#Inv.	#Hot	Hot KB
avrora	1.88	139	3,481	71,914	66,896	223,645,512	3,808	630	894
batik	2.27	145	5,106	200,276	198,085	329,194,041	8,073	1,451	2,357
eclipse	21.22	2,135	9,157	5,059,544	542,729	5,261,032,053	16,785	5,446	7,177
fop	1.56	53	10,234	139,724	138,390	94,280,502	7,450	1,573	3,118
h2	3.90	2,597	3,367	3,735,070	736,734	2,462,413,765	4,804	1,093	2,113
jython	5.81	729	10,903	2,181,187	535,963	1,871,017,740	9,100	2,226	5,444
luindex	1.03	100	2,973	42,935	34,224	242,376,049	3,476	532	1,102
lusearch	2.83	310	2,633	5,209,205	490,887	1,332,093,822	2,901	495	894
pmd	2.22	143	3,522	361,285	265,307	224,140,701	5,661	1,758	2,982
sunflow	2.14	1,446	3,900	2,471,177	511,412	1,828,971,660	4,457	405	1,107
tomcat	3.21	184	8,520	633,851	315,908	599,033,048	13,465	3,092	6,465
tradebeans	3.18	-	-	-	-	-	33,653	3,055	5,868
tradesoap	10.30	-	-	-	-	-	34,319	6,044	12,470
xalan	2.23	268	3,590	936,435	475,789	620,287,563	4,815	1,820	3,197

Our evaluation reuses the collected profile data to produce the effect of each partitioning strategy. At each profile interval, we compute a partitioning of the application's allocation sites using one of the strategies described below. Then, we assign the interval's object size and access counts as either "hot" or "cold" according to the partitioning. If the interval contains an allocation site that is not in the partitioning, we always opt to assign the unknown allocation site's size and access counts to the "hot" set.

Our study compares the following partitioning strategies:

Future:. Our *baseline* strategy uses future memory usage data to achieve the *ideal* hot/cold partitioning. It recomputes the knapsack partitioning at the start of every profile interval using the size and access counts for the *same* interval.

Offline:. An offline strategy only computes the knapsack once using aggregated size and access counts from all the profile intervals in a separate run. We evaluate two different offline strategies: (a) **Offline-same:** The same program input is used to generate the offline knapsack and evaluate the heap partitions, and (b) **Offline-diff:** The offline knapsack is generated and evaluated using a different program input. Specifically, DaCapo's small input size is used to compute the offline knapsack when evaluating heap partitions for a run with the default input.

Reactive:. The reactive strategies use past profiling information from the same run to guide partitioning at each interval. We evaluate two different sets of reactive strategies: (a) **Reactive-KS:** These techniques recompute the knapsack at every interval using aggregated profile information from the interval(s) immediately preceding the current interval, and (b) **Reactive-fast:** It is often infeasible to construct the knapsack partitioning at run-time due to the large overhead of profiling object access patterns. Our earlier work [Jantz et al. 2015] provides a *realistic* reactive approach based on the low-overhead scheme proposed by Huang et al. [Huang et al. 2004]. This realistic reactive online profiling approach samples thread call stacks to construct its profile input. For our model, call stacks are sampled once every 10msec, for a total of 25 samples per interval with our default heap sampling rate of 250msec. For both reactive strategies, we evaluate five distinct configurations that differ by the number of profile intervals they use to construct the partitioning scheme (either 1, 2, 4, 8, or all of the preceding intervals).

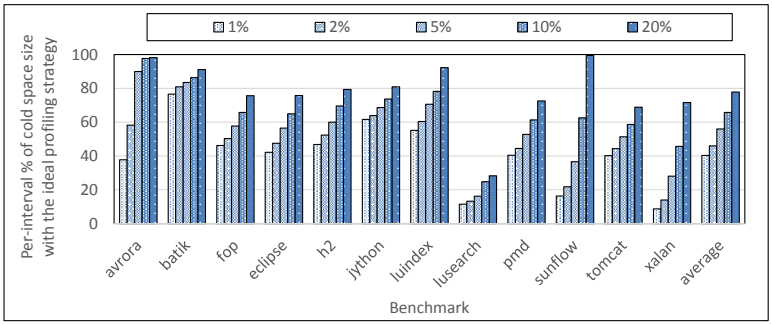

Fig. 2. *Future* profiling (an idealized approach for baseline comparison) effectively finds a high fraction of 'cold' heap objects that account for a small fraction of total heap accesses (given by the knapsack capacity).

4.3. Results and Observations

Cold Space Sizes with Future Profiling. Figure 2 shows, for each benchmark, the portion of objects assigned to the cold space with *future profiling* with different knapsack sizes. As expected, increasing the knapsack capacity allows the approach to add more objects to the cold space. On average, future profiling policy identifies 40.3% of heap objects that account for 1% of heap accesses, and 77.8% of objects that account for up to 20% of heap accesses. The size of the cold space objects varies significantly among different benchmarks. For instance, with *sunflow*, the coldest objects that account for up to 20% of heap accesses make up more than 99% of the heap, while, with *lusearch*, the same size knapsack yields a cold set with only 28% of total heap space.

Accuracy of Different Profiling Strategies. Figures 3 and 4 summarize the accuracy of the hot/cold partitionings generated with each profiling strategy *in terms of misassigned heap accesses and size compared to the future strategy*, respectively. For each profiling strategy, the positive bar shows the accumulated heap accesses/size from allocation sites that are predicted to be cold by the profiling strategy, but are determined to be hot by the *future* strategy, while the negative bar shows the accumulated result from sites that are predicted to be hot, but are assigned as cold by the *future* strategy. Other than the reactive-fast policy, we evaluate each strategy with five knapsack capacities (1%, 2%, 5%, 10%, 20%), and compare it against the *future* configuration with the same size knapsack. Although the reactive-fast strategy does not employ a knapsack partitioning, we still present five sets of bars to show its accuracy relative to the *future* policy with different knapsack sizes. Results are computed at each program interval, normalized by the total amount of heap accesses/size at the interval, and then averaged over all intervals. Due to space constraints, we plot only one pair of bars for each of the offline strategies and five pairs of bars for each of the reactive strategies. The sets of bars for the reactive-KS and reactive-fast strategies are plotted in ascending order by the number of preceding intervals of profile data that were used to construct the partitioning strategy (i.e., from left-to-right, the bars show results computed with 1, 2, 4, 8, and all preceding intervals of profile data).

Object partitioning affects both heap access counts and size, and makes it difficult to compare profiling strategies that result in different allocations of both metrics. To directly compare each strategy, we develop an approach that uses *fixed* heap sizes at each interval. These experiments employ the same profiling strategies, but modify the partitioning routines so that the sizes of the hot/cold sets match (or are as close as possible to) the sizes generated by the future strategy *for the same program interval*. We first compute an ideal partitioning where, similar to our earlier experiments, the percentage of accesses to the cold set is as close as possible to the knapsack size, without

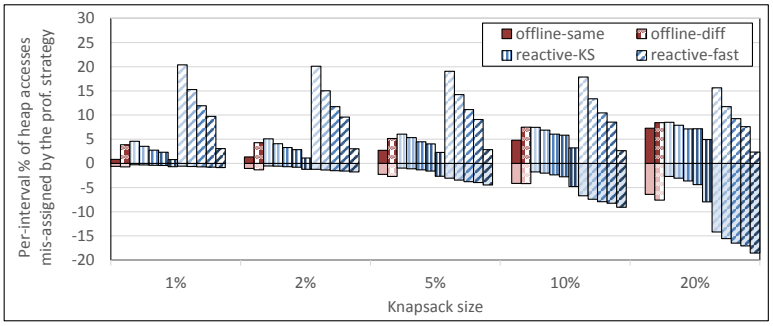

Fig. 3. Mis-assigned heap accesses compared to future profiling. Standard profiling strategies mis-assign a large fraction of heap accesses to the cold (positive bars) or hot sets (negative bars). Mis-assigned heap accesses to the cold space can quickly negate the energy saving potential of the heap memory management.

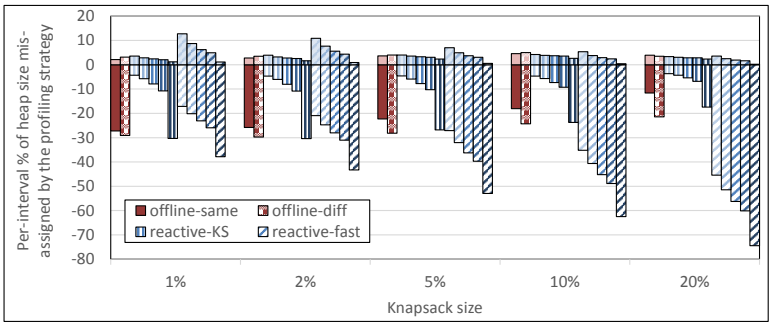

Fig. 4. Mis-assigned portion of object size compared to future profiling. Standard profiling strategies mis-assign a fraction of hot heap objects to the cold set (positive bars) and cold heap objects to the hot set (negative bars). Mis-assigned cold objects indicate a sub-optimal utilization of the low-power (cold) DIMMs.

exceeding it. Next, the modified knapsack routines, given the ideal sizes, partition the objects so as to maximize the amount of accesses to the hot set. For the reactive-fast strategy, if the generated hot and cold sets do not match the ideal sizes, allocation sites are randomly added to (or removed from) the hot set until the hot set size is equal to or just less than the ideal size. At each interval, the modified strategies employ *future* knowledge of the ideal hot and cold heap sizes. Hence, these experiments are not intended to estimate the effect of each strategy in a realistic system, but are only used to ease the comparison of the relative accuracy of each profiling approach. Figure 5 shows the percentage of accesses mis-assigned to the cold set with the modified partitioning strategies with fixed heap sizes. This experiment provides a sense of the relative energy benefits that can be achieved by each profiling strategy for this adaptive VM task.

Together, Figures 3, 4, and 5 allow us to make a number of interesting observations. (a) Intrinsic limitations prevent even the *best* offline and online profiling strategies from achieving the effectiveness of the *future* policy for this heap memory management optimization. The mispredictions seen by the offline-same strategy show the effect of profile data aggregation. The reactive-KS policy degrades performance (over *future* strategy) because past program behavior is not always able to accurately predict the future execution. (b) The accuracy of the offline strategy depends on the quality of its training data. For instance, in Figure 3, observe that, for each knapsack capacity, the offline-diff approach generates a larger portion of mis-assigned heap accesses than the offline-same strategy. (c) Figure 3 also shows that the reactive strategies that use only the most recent profile interval mis-assign more heap accesses than configurations

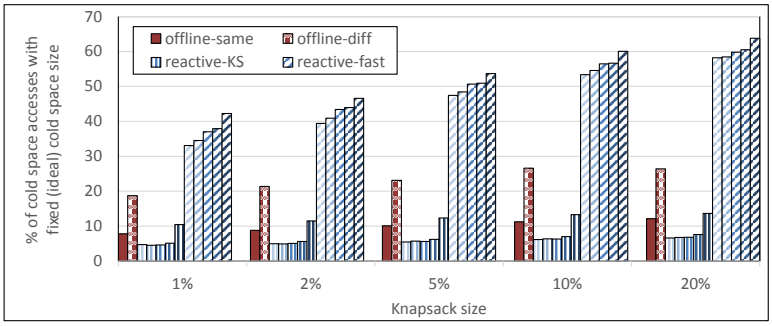

Fig. 5. The fixed heap-size configuration allows a more direct comparison of the accuracy (effectiveness) of the different profiling strategies for heap memory management.

that use data over more past intervals. However, Figure 4 reveals that profiles constructed with older information tend to mis-assign a larger portion of heap object data. We suspect this effect is due to the fact that recently-used objects are more likely to be accessed than objects used in the distant past, and so, profiles that give equal weight to both tend to assign more objects to the hot space. Figure 5 shows that, overall, only using the more recent profile data during online profiling seems to be a more effective policy for this FDO. (d) For both heap accesses and size, employing additional past profile intervals with the reactive-KS strategies seems to mirror the aggregating effect noticed by the offline policies. (e) Although the reactive-KS strategy is more effective, it requires a very high implementation overhead. Unfortunately, the more practical reactive-fast strategy, which relies on partial profile data and a simpler partitioning scheme, is not sufficiently accurate. This observation suggests the need for further research to achieve the potential of online profiling in standard runtime systems.

Impact of Profiling Frequency. All of the preceding experiments profile memory activity at the same rate (of 250msec). To understand how different profiling rates affect our results, we collected heap usage activity for each benchmark with five distinct profiling rates: 50msec, 100msec, 250msec, 500msec, and 1 second. For each selected heap profiling rate, the reactive-fast strategy uses the same call stack sampling rate (of 10msec). Although the standard sampling mechanism in HotSpot supports a minimum sampling rate of 10msec, we selected 50msec as the smallest profiling interval because: 1) we wanted to ensure that the reactive-fast strategy had more than a few stack samples per interval, and 2) we found that shorter rates required prohibitively large space and time costs. The maximum rate of 1 second was selected to ensure that each of our benchmarks produces more than a few profile intervals.[5]

Figure 6, which is presented in a similar style as Figure 3, shows the percentage of mis-assigned heap accesses for each profiling strategy with a 5% knapsack with different profiling rates.[6] For the profiling intervals we tested, the interval length only has a small effect on the accuracy of most profiling strategies. The exception is the reactive-fast strategy, which becomes more accurate as the length of the profiling interval is increased. Since the stack sampling rate is kept constant, a longer profiling interval allows the stack profiler to collect more samples during each heap profiling interval. Thus, with the reactive-fast strategy, it is important to balance the frequency and overhead of stack sampling to achieve the best possible accuracy and performance.

[5]The smallest benchmark in our set, *fop*, generates 13 intervals with a 1 second profiling rate.
[6]Due to space limitations, we do not present heap size results for different profiling rates. However, the trends found in the size results are similar to Figure 6, and do not change our conclusions.

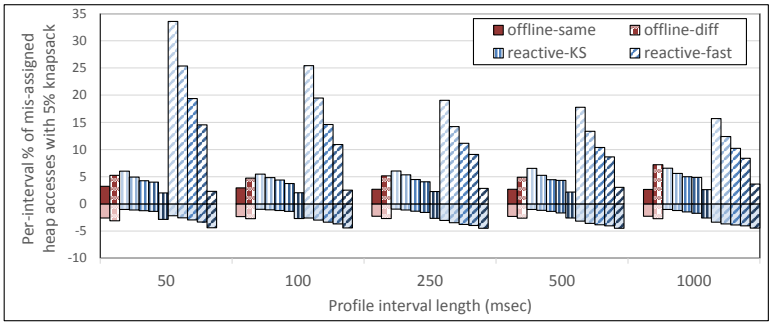

Fig. 6. In theory, more samples (shorter intervals) allow us to better understand the intrinsic data aggregation effect for the offline strategies and the impact of past (rather than future) profiles for the online strategies. However, shorter intervals diminish the accuracy of the *reactive-fast* strategy because it uses a constant call-stack sampling rate (of 10msec) and collects fewer samples per interval.

5. CODE CACHE MANAGEMENT

We use code cache management (CCM) in run-time systems as our next adaptive profile-driven VM task. We study how data obtained from offline, online-reactive, and default profiling mechanisms can impact the performance of the CCM algorithm compared to the baseline *future profiling* at different constrained code cache sizes. Our experiments model the *client* HotSpot VM configuration that includes the interpreter and one compilation level (c1). We choose this HotSpot configuration because by using a lower compilation threshold it may place greater pressure on the code cache by compiling more methods at program startup.

The code cache storage enables the native code produced for a method after JIT compilation to be reused later, without re-generating it on every invocation. The code cache management in the VM is responsible for finding and evicting previously compiled regions from the cache, (a) to maintain program correctness in dynamic languages if the assumptions made during compilation are later found to be incorrect, and (b) to make room for the native code from later compilations if the code cache is full.

The code cache is a constrained resource. VMs can place a heavy load on a system's memory resources, especially on memory-constrained devices. We found that compiling just the hot program methods (with the c1 compiler) for the DaCapo benchmarks results in an average code cache size of over 4MB. Devices also typically have multiple processes running simultaneously. Thus, performance efficiency with a small code cache is an important goal for systems operating with resource constrains.

The CCM algorithm has a choice when selecting a method to purge from the code cache to accommodate a later compilation. Ideally, the algorithm needs to find a method that is not currently hot and will not become hot in the future. Better code cache management can enable the VM to enhance performance by supporting larger applications, allowing more aggressive compilation to improve performance, and by keeping more programs simultaneously resident in memory to improve response time.

5.1. Performance Metric

Similar to our methodology in the last section, we construct a trace-based experimental setup to explore the impact of the limitations of different profiling strategies on the effectiveness of the CCM algorithm. Our experimental setup is described in Section 5.2. Here, we describe the performance metric we devise for our CCM experiments.

Method Hotness Count:. JIT compilation attempts to improve performance by reducing the amount of time spent by the program in the slower execution (interpretation)

mode. The profiler in the HotSpot interpreter uses the method's *hotness_count* to estimate the time spent in the method. Thus, a lower total hotness_count over all program methods indicates that the program spent less time in the interpreter and more time in high-performance compiled native code, which should result in better performance.

If a previously compiled method is evicted from the code cache, then future invocations of the method will execute in the interpreter, until the evicted method becomes hot again and is recompiled. Thus, on every request to create space for a new method compile, a good CCM algorithm should find a method to evict that minimizes the (future) time spent by the program in the interpreter. Hence, better code cache management will result in a smaller total program hotness_count over the entire program run. Our framework computes the *total program hotness_count* as the primary measure of the quality of the CCM algorithm.

From Hotness Counts to Execution Time:. In addition to the hotness_count, it is interesting to study the effect of different CCM policies on program *execution time*. We develop a simple mechanism to associate program hotness_count with program execution time. Such association depends and is specific to the characteristics of the selected VM (and interpreter/compiler). In this section we present our mechanism to achieve this association for the HotSpot VM using both its c1 and c2 JIT compilers.

To relate hotness_counts with program run-time we execute each benchmark with the default HotSpot VM (for both c1 and c2 separately) for 24 different hotness thresholds from 100 to 100,000,000. Each benchmark and hotness threshold configuration will send a different set of methods to compile in the first program *iteration*. We run each benchmark for 10 iterations. We only allow methods selected for compilation in the first iteration to be compiled. We extract the total program hotness_count and execution time (program wall-time) for the last iteration in each run to discard the compilation overhead. We then plot all of the points associating hotness_count and run-time for each benchmark, and then fit a (quadratic) curve over these points.

We use the developmental release of HotSpot/JDK-9 to conduct our experiments to associate program hotness_count with execution time. HotSpot-9 implements a *segmented* code cache that make it easier and more precise to control the per-benchmark size of the code cache. Our experiments associating hotness_counts to execution time employ 12 DaCapo Java benchmarks with their *default* input size. [7]

Figure 7 shows these association plots for the different DaCapo benchmarks with both the c1 and c2 compiler configurations. Thus, we can see that (for the HotSpot VM and DaCapo benchmarks) interpreter hotness_counts are a good indicator of overall program performance, even when the measured execution time includes aspects of VM execution such as CCM and garbage collection. The per-benchmark mathematical equation forming the regression curve is used to associate hotness counts with *time* during later experimental runs. We employ both the hotness_count and (correlated) execution time to compare different profiling policies in the remainder of this section.

5.2. Experimental Setup

Methodology: We instrument HotSpot in JDK-6 to generate and log the trace and execution data for our experiments. We conduct two runs for each benchmark. In the first run, HotSpot runs the program in the interpreter alone, and divides the execution into 10msec intervals.[8] At the end of each 10msec interval, HotSpot dumps the

[7] *batik*, and *eclipse* fail with the *default* build of HotSpot-9 without any of our modifications. *batik* seems to fail due to updated reflection rules in HotSpot-9 that prevent access to some JVM internal packages (under com.sun.*). *eclipse* fails with the error message "The type java.lang.CharSequence cannot be resolved".

[8] We use 10msec intervals since we found that to be the smallest stable sampling interval for HotSpot's monitoring subsystem.

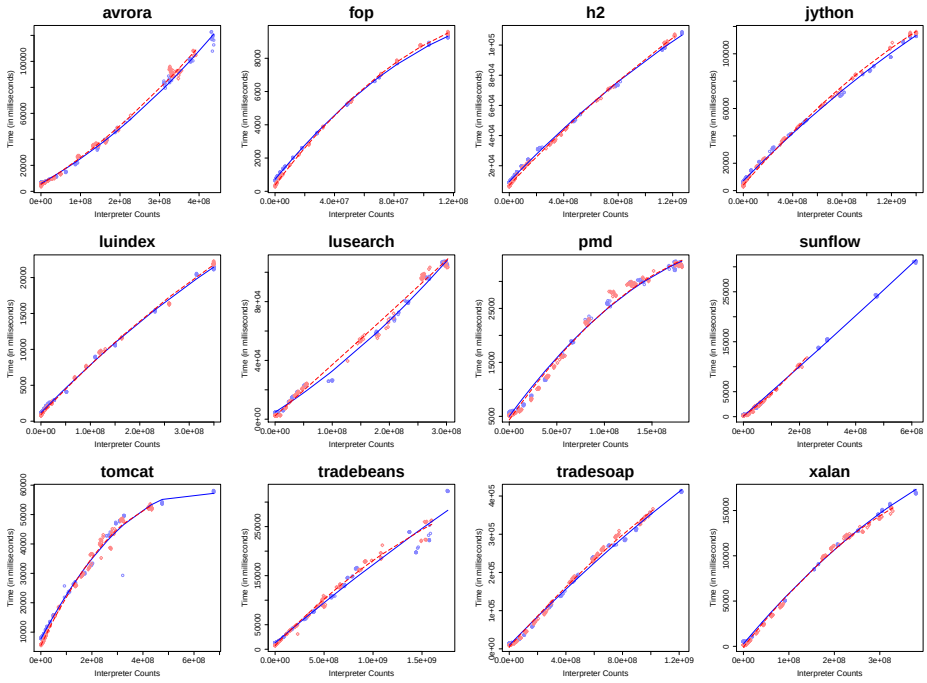

Fig. 7. These plots provide the equations associating hotness counts with program execution time. For each program run, the corresponding interpreter count and run-time (in msec) are plotted on the X-axis and Y-axis respectively. We use the solid (blue) line for the 'c1' plot and the dashed (red) line for the 'c2' plot. As expected, program execution time with the 'c2' compiler is typically faster and with a smaller hotness_count (likely due to more inlining) than that with the 'c1' compiler for a given compile-threshold value.

hotness_counts of all program methods. The other program run determines the size of the compiled native code with the client build of HotSpot-6 for hot program methods. For each benchmark, we also measure the maximum space needed for the code cache when all hot methods are compiled and resident in the cache.

Our evaluation runs use this trace data to reproduce the operation of the code cache manager with different method eviction algorithms and code cache sizes. Experiments use 100%, 90%, 75%, 50%, and 25% of the maximum code cache space needed for each benchmark. The maximum code cache space is the total accumulated size of all compiled methods in the default program run, and is different for each benchmark. Table I presents the maximum code cache space, as well as the total number of invoked and hot methods (with c1's default compile threshold), for each of the DaCapo benchmarks.

At the end of each 10msec interval, a method is compiled if its total hotness_count exceeds the default HotSpot compilation threshold. If the code cache is full, then the code cache manager uses one of several strategies to find and evict existing methods from the code cache. On every eviction request, each algorithm finds contiguous space that is equal to or greater than the size of the new compiled method. If the new method does not occupy the entire space that is created, then the remainder can be merged with the adjacent unoccupied blocks, whenever possible. We experimented with the following method eviction algorithms:

Future:. Our *baseline* heuristic looks into the *future* profile of the program to find close to the best set of contiguous methods to evict from the code cache to fit the new compiled method. It finds the set of methods that, combined together as a unit,

have the smallest remaining hotness_counts. Thus, with this algorithm, methods that will never be used again are given the highest priority for deletion, and are sorted based on their size (largest size first). Methods that will never be compiled again are given the second highest priority and will be deleted in the order of their future hotness_counts (fewer counts first). Lowest priority is given to methods that will exceed their compile threshold again, sorted to order later compiles first.[9]

Offline:. This set of algorithms use information from a prior program run, and aggregate the information over all intervals of the profile run. The profile data is used to sort methods in ascending order of their total hotness_counts over the entire run. Then, in the later measured run, methods are selected for eviction from the code cache in the order of lowest counts first. We study the following offline profiling schemes: (a) **Offline-same:** The same input is used for the offline profiling run and the later evaluation/measured run. (b) **Offline-diff:** The profiling run uses the DaCapo *small* input while the measured runs uses the *default* input. With different inputs for the profiling and measured runs, it is possible for the profile to not have any information about certain events (invoked methods) in the measured run. For such methods, this algorithm assigns the lowest priority for eviction.

Reactive:. For these CCM algorithms, profiling data collected during the past execution of the same program run is used to guide the CCM task to optimize the remaining program execution. In this case the best (set of contiguous) methods to evict is determined based on their hotness_count in earlier intervals of the same run. The following formula finds the hotness_count for each method by assigning progressively lower weights to older profile data: $\tau_{n+1} = \alpha * t_n + (1 - \alpha)\tau_n$ where, τ_{n+1} is the predicted hotness_count for the next interval, and t_n is the actual hotness_count in interval 'n'. We experimented with α values of 0.1, 0.3, 0.5, 0.7, 0.9, and 1.0. We present the results for $\alpha = 0.1$, which provided the best overall numbers.

Stack scan:. The default policy in the latest HotSpot-8 release (that we call, *stack_scan*) uses low-overhead sampling based profiling. Stack_scan uses a *sweeper* thread to periodically mark the methods found on the call-stack of any application thread. This policy evicts a method from the code cache if it is left unmarked over several sweeper periods. This policy disables compilation if the code cache is full, and restarts compilation after the sweeper again creates adequate free space in the code cache. We implement a version of this algorithm in our trace-based experimental framework as an example of a *practical* online/reactive CCM policy.

For our experiments, all compiled methods are eligible to be de-compiled. Additionally, all methods marked for eviction are deleted instantaneously from the code cache. Thus, a method evicted from the code cache will be interpreted in the next interval.

5.3. Results and Observations

In this section we present the results of our experiments to evaluate the effectiveness of different CCM algorithms compared with our baseline *future profiling* approach that uses knowledge of the future program behavior to make CCM decisions.

Performance Potential with Future CCM Policy. Figure 8 shows the potential of *future profiling* with CCM at different constrained code cache sizes. Along the primary Y-axis, each bar plots the ratio of the program hotness_count with the *future* CCM

[9]Our *future profiling* CCM heuristic is not guaranteed to produce *optimal* results. An ideal CCM strategy will require solving the knapsack problem while allowing *fragmentation* in the code cache or a complete reorganization of the code cache at each (10ms) interval. Since (as opposed to our approach in Section 4) the future profiling decisions are not used to measure the performance of the other CCM techniques (*hotness_count* is used instead), we decided to employ a simpler *future* CCM algorithm that works well in practice.

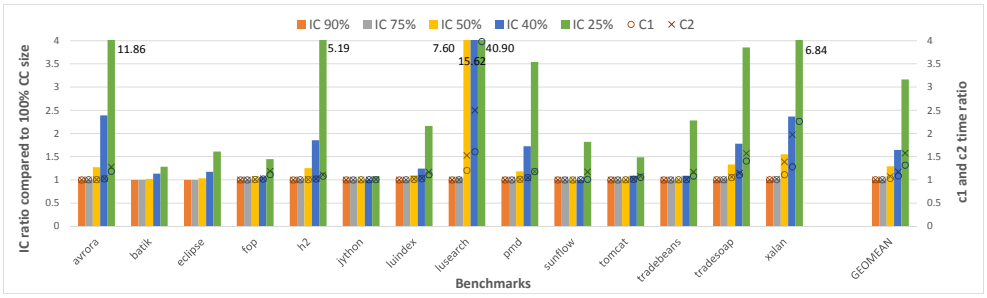

Fig. 8. Using the *future* program execution profiles allows the CCM algorithm to often find the right methods to evict from the code cache and minimize the performance impact of a constrained code cache size.

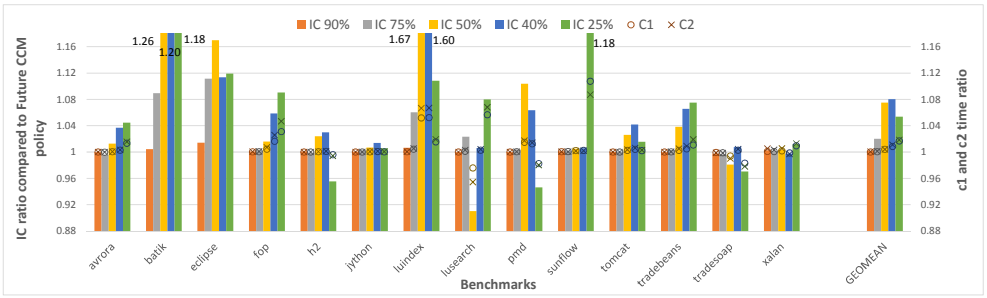

Fig. 9. Past execution profiles (with no cost/overhead concerns) allow CCM algorithm using the *reactive* profiling strategy to deliver performance that is very close to that achieved with the *future profiling* baseline.

policy and indicated code cache size to the run-time with the same algorithm and an unlimited code cache. An unlimited code cache never needs to evict compiled methods from the cache. We observe that the *future* CCM algorithm often finds the right methods to evict from the cache to minimize performance impact. On average, we see very negligible performance losses with code cache sizes restricted to 90%, and 75% of required code cache space. With 50%, 40% and 25% of desired code cache size the (geometric) average hotness_count loss is 29%, 64% and 3.16X respectively. The corresponding impact in terms of (correlated) program run-time with the HotSpot c1 (circle) and c2 (cross) JIT compilers is indicated along the secondary Y-axis in Figure 8. This result shows that CCM has the potential to significantly reduce an executing program's code cache memory requirement with small performance losses in many cases.

Performance Potential of Other CCM Policies. Next we compare the performance effectiveness of the other CCM policies as compared to the performance delivered by the future profiling baseline. The profiling driven CCM algorithms in our framework have access to the most comprehensive, accurate, and timely profile data possible by that profiling technique with no run-time overhead.

Figure 9 shows the performance of the CCM algorithm when using the best *Reactive* profiling strategy (for $\alpha = 0.1$) as *compared with the corresponding future profiling baseline and corresponding code cache sizes*. Again, the hotness_count ratios are plotted along the primary Y-axis and the corresponding correlated program run-time ratios with the c1/c2 compilers are shown on the right Y-axis. We find that a good reactive strategy can achieve program performance close to that delivered by the *future* policy even for heavily constrained code cache sizes. The average hotness_count degradation (compared to the *future* policy) with this reactive strategy are only 0.2%, 2.0%, 7.5%,

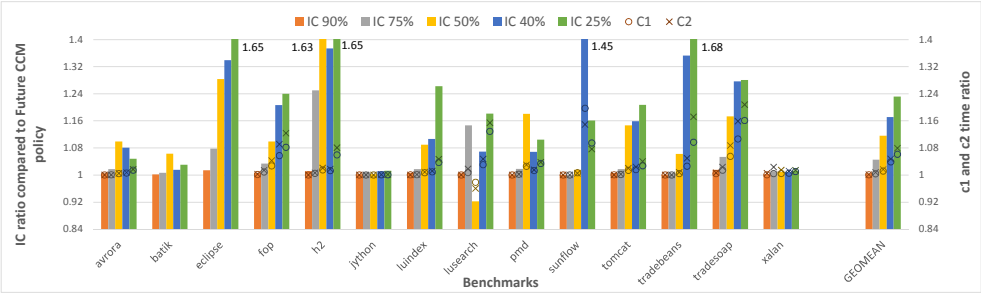

Fig. 10. The intrinsic profile data aggregation effect during the *offline-same* profiling strategy lowers the effectiveness of the CCM algorithm.

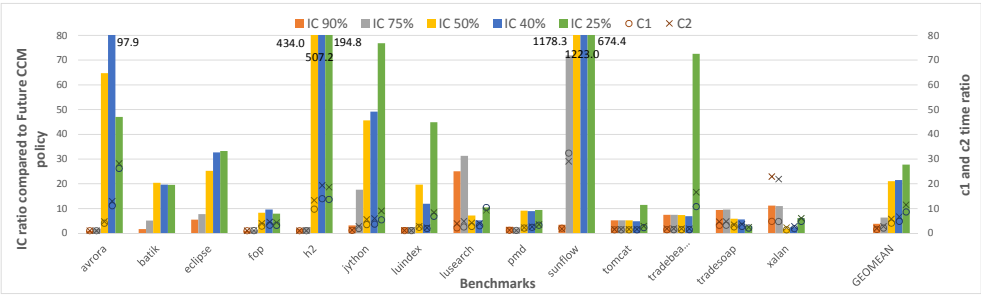

Fig. 11. *Practical* reactive profiling strategies (like *stack-scan*) used by VMs to manage run-time profiling overhead are unable to realize the potential of the *reactive* strategy for CCM.

8.0%, and 5.4% for code cache sizes that are 90%, 75%, 50%, 40%, and 25% of the maximum needed, respectively. These results suggest that past program behavior is a good indicator of future execution for CCM for our benchmarks. Remember that the cost of collecting profiling information at run-time is ignored during this algorithm.

Figure 10 presents the performance comparison of the *offline-same* code cache eviction algorithm compared with the corresponding *future* policy. We see that with a perfectly representative offline profile, the CCM algorithm performs quite well. The *offline-same* strategy results in an average hotness_count loss of 0.5%, 4.5%, 11.6% 17.1%, and 23.2% for our five code cache sizes respectively, compared to the *future* algorithm. Even with perfect offline profile data, the performance loss with *offline-same* is higher than that observed with the *reactive* CCM strategy, and indicates the negative impact of profile data aggregation during offline profiling for CCM.

Again, we see a larger performance loss when the profile does not (exactly) match the measured run. We find performance (hotness_count) losses of 5.2%, 22.1%, 60.0%, 84.7%, and 90.6%, on average, with the *Offline-diff* scheme compared to the *future* strategy for the code cache sizes of 90%, 75%, 50%, 40%, and 25% respectively.

Performance with Default CCM Policy. HotSpot uses a low-overhead sampling-based mechanism to collect partial profiling data to guide its default *stack-scan* CCM policy. We realize that this default HotSpot CCM policy may not be tested or intended to be employed on devices with constrained code caches. Yet, we employ this CCM strategy as an instance of a fast, low-cost and less precise online reactive profiling policy. The actual implementation of this policy in HotSpot has been heavily tuned for different situations, and is associated with several flags and other tuning knobs. We implemented a close variant of this complex policy in our framework.

Figure 11 displays the the hotness_count (along the primary Y-axis) and c1/c2 correlated time (along the secondary Y-axis) comparison of the *stack-scan* CCM algorithm compared with the corresponding *future* CCM approach. We found that this policy fares quite poorly and achieves performance (hotness_count) that is 3.79X, 6.40X, 21.07X, 21.57X, and 27.80X worse over the *future* configuration, on average, at 90%, 75%, 50%, 40%, and 25% code cache sizes respectively. This result demonstrates the importance of profiling accuracy, and suggests that imprecise profiling may misrepresent the true worth of adaptive optimizations and even prevent their adoption.[10]

6. SELECTIVE JIT COMPILATION

The final adaptive optimization task we study is *selective compilation*. This technique finds and compiles only the important (or *hot*) program methods to limit compilation overhead while maximizing the overall performance benefit [Hölzle and Ungar 1996; Paleczny et al. 2001; Krintz et al. 2000; Arnold et al. 2005]. In HotSpot, a method is hot if its *hotness_count* exceeds a fixed threshold value. Selective compilation needs accurate profile information about the hot program methods to be most effective.

We choose to study selective compilation in this work as it provides an example of a task that differs from the adaptive VM tasks previously studied in this paper. First, (given a large enough code cache) decisions for selective compilation are only taken once and use aggregate method hotness information over the entire program run. Therefore, data aggregation is a non-issue during the offline profiling strategy for (the HotSpot implementation of) selective compilation. Second, fast and low-overhead online reactive profiling to find the set of hot methods for selective compilation is feasible with current VM technology. Therefore, we do not develop or investigate any other *practical* implementation of online/reactive profiling for selective compilation (as we did for the previous adaptive tasks studied in this paper).

Our experiments in this section use program execution times from actual program runs in HotSpot. We extend HotSpot as described below to accomplish selective compilation for all our profiling strategies. We conduct this study in HotSpot-6 using the default c1 (client) compiler. Our run-time experiments were performed on a cluster of 8-core 2.83GHz Intel x86 machines.

6.1. Experimental Approach

We evaluate the following profiling strategies for selective compilation.

Steady:. Our *steady* configuration assumes that all hot methods are accurately known and compiled *before* the program begins. We use this configuration to provide a lower-bound for the program execution time.

Future:. For VMs that support *background compilation* [Krintz et al. 2000], the order of compiling program methods can affect the program run-time. The future profiling baseline can employ knowledge regarding the future program execution to compile the set of hot methods in the *order* that will minimize the overall time spent by the execution in the slow interpreter mode (and maximize the time spent in JIT compiled code) [Ding et al. 2014]. Although HotSpot supports background compilation, we disable this feature for our present study to nullify the effects of compilation delay due to queue backup at program startup [Jantz and Kulkarni 2013], which can conceal the effects that we actually wish to investigate for this work. Consequently,

[10]Since this CCM policy is available in HotSpot, we conducted actual HotSpot runs with the same constrained code cache sizes. We used HotSpot-9's client (c1) compiler, and a *startup* configuration where each benchmark was run for one iteration. With these actual HotSpot runs, the default *stack_scan* policy results in an (geometric) average performance (execution time) loss of 30%, 43.8%, 2.5X, 3.6X, and 5.7X at our five code cache sizes respectively compared to a HotSpot configuration with unlimited code cache size.

the future strategy now mirrors the offline-same configuration for this study. Additionally, disabling background compilation allows us to more precisely account for the compilation overhead, and allow compiled methods to become available earlier.

Offline-same:. Our offline-same configuration employs a profile/training run to determine the set of hot methods using the same program input as that used later during the measured evaluation run. During the evaluation run, this known set of hot methods are sent to compile in the order the methods are reached during execution. The hot methods are compiled on their first invocation.

Offline-diff:. For offline-diff, we again employ the results of the profile run with the *small* input for an evaluation run with the *default* input.

Reactive:. This is the default profiling policy used by HotSpot. Each method is compiled if and when it reaches the compile threshold. A reactive strategy works on the assumption that a method detected as hot in the past program run will remain hot in the future. In cases when this assumption is incorrect, a reactive profiling based configuration may lose performance due to the inability to recoup the compilation overhead by faster program execution in the resulting generated native code.

Several implementation issues need to be overcome in HotSpot to enable a fair comparison of different profiling strategies. Such issues include: (a) We need the ability to ignore the profiling and compilation cost for our *Steady* strategy. (b) Each strategy should compile the same set of hot methods for each benchmark-input *configuration*. (c) Varying method compilation orders can influence performance by affecting optimizations, especially method *inlining*. (d) We need the capability to force HotSpot to compile the pre-determined set of hot methods. For instance, with the *Offline-diff* policy, program runs with the *default* input should compile methods found to be hot during the program run with the *small* inputs. (e) All strategies, other then *Reactive*, compile their methods at the earliest opportunity (execution count of 1). HotSpot implements a few FDOs in the *c1* compiler, that rely and work differently depending on the values in the method's invocation, backedge, and other profiling counters. Therefore, these counter values need to be warmed-up (about equally) for all our experiments.

We update HotSpot to resolve these issues for this study. We configure the DaCapo benchmark *harness* to run each program over multiple *iterations*. We add a new set of per-iteration method counters in HotSpot that are incremented during the run, but are reset after each iteration. The first iteration performs no compilation and is only used to *warm* all method counters. HotSpot detects the end of each benchmark iteration using our added callback. At the end of the first iteration, HotSpot loads a file containing a pre-determined set of selected hot methods to compile, and marks them. The next iteration compiles the marked methods when the per-iteration method counters reach their desired thresholds. The execution time of this iteration is recorded as the program run-time, and is used as our performance metric in this section. For the *steady* policy, the benchmark is run for a few more iterations before recording the run-time to discard the compilation overhead.

To limit the impact of different method compilation orders, we currently turn off method inlining for the *Offline* and *Reactive* profiling strategies. We also disable HotSpot's OSR (on-stack replacement) compilation to ensure that method compilation proceeds uniformly in all our configurations. Thus, while the exact implementation details vary, our framework is similar to the widely used experimental methodology called *replay compilation* [Huang et al. 2004] that is designed to facilitate performance analysis by controlling nondeterminism during different runs and configurations.

Results and Observations. Figure 12 compares benchmark run-times with our selected profiling strategies for selective compilation. To account for inherent run-time timing variations, all our execution-time results report the mean and 95% confidence

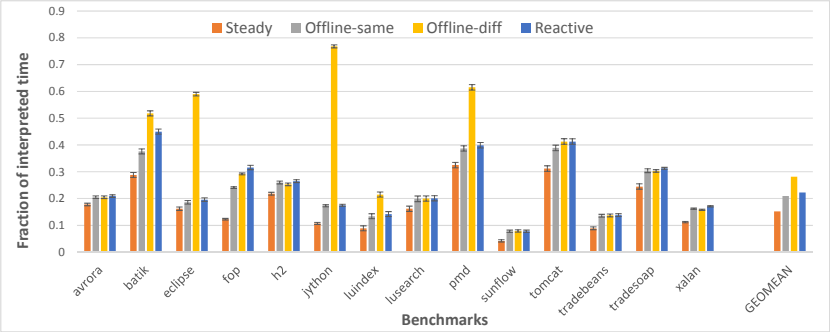

Fig. 12. Absence of the profile data aggregation effect allows the *offline-same* profiling strategy to exceed the performance achieved by the *reactive* strategy, which suffers due to a delay in collecting and utilizing the profile data at run-time, during selective compilation.

intervals over 10 runs for each benchmark-configuration pair [Georges et al. 2007]. Our observations reflect the differences identified earlier between selective compilation and the previously studied adaptive VM tasks. The baseline employs the **Interpreted** strategy, where no methods are compiled and the program runs in the interpreted mode for its entire execution. We again find that *representative* program inputs are important during offline profiling. Since offline profiling mechanism does not suffer from the effect of aggregating profile data, it can be more effective than the *reactive* scheme for selective compilation. Likewise, the reactive mechanism does not suffer from a high online profiling penalty. However, compilation decisions are delayed until sufficient profile data is available after program start, which forces the execution to spend more time in slow interpretation mode. This delay penalizes performance compared to *offline-same*, and has been reported by some previous studies [Kulkarni 2011]. On average, we see that program performance with *offline-same*, *offline-diff*, and *reactive* profiling policies are 38%, 85% and 46% worse than the *steady* policy respectively.

7. DISCUSSION AND FUTURE WORK

In this work, we design and build innovative experimental frameworks to understand the fundamental limitations of offline and online profiling techniques. We quantify the impact of these profiling limitations on the effectiveness of multiple real adaptive VM tasks, as compared to a constructed baseline that knows the precise relevant future program behavior. We design the future, offline, and reactive strategies to assume no profiling overhead and the profiles generated to be as accurate as can be obtained by each technique. Additionally, we compare our results from these *idealistic* models to profiling techniques that have been used or proposed for a *realistic* VM setting.

Our study performs a systematic evaluation and quantitatively confirms many known or expected issues with online and offline profiling for the adaptive tasks investigated and benchmarks used. Our experiments enable us to make the following observations. (1) Experiments using our *future profiling* baseline identify the true potential of VM optimization tasks. Knowledge of the ideal impact is critical to prevent the possibility of an adaptive task being discarded due to poor performance resulting from inaccurate or partial profile data. (2) We validate that intrinsic profiling limitations have a noticeable impact for many adaptive VM tasks as compared to the future profiling baseline. Intrinsic limitations include profile data aggregation during *offline-same* profiling and the reliance on past, rather than the future, program behavior knowledge during *online-reactive* profiling. (3) We found that past profile is an excellent predictor of future program execution behavior for the adaptive tasks and

runtime employed. Accurate and comprehensive past profiles can enable the adaptive VM task to attain close to ideal benefits, but may be too expensive to obtain at runtime. At the same time, incomplete online profile data was seen to severely reduce the effectiveness for the studied FDOs. This justifies the need to continue enhancing the standard profiling techniques used in current runtime systems to enable reactive profiling based adaptive tasks to reach their performance potential. (4) FDOs relying on offline profiling show noticeable negative performance effects from profile data aggregation. However, matching profiles used during the training and evaluation runs can still enable excellent performance for dependent FDOs. The challenge then is to devise techniques to find such matching profiles for *all* program executions irrespective of input and environmental settings. (5) As expected, dissimilar input sets used during the training and evaluation program runs (*offline-diff* configuration) were seen to significantly reduce FDO performance. Yet, in many cases, offline-diff still achieves higher effectiveness than the *practical* reactive techniques that only provide access to lossy profile data due to the issues of run-time overhead and profile data collection delay.

Accurate online profiling can involve an intolerably high overhead at run-time. For instance, the reactive strategy we employ for CCM will require the profiler to insert a counter at each function entry and loop back-edge. Likewise, naïvely profiling object accesses during heap memory management will also likely cause unacceptably high overheads. Researchers are developing mechanisms to increase accuracy of online profiling while reducing overhead using unutilized hardware [Yang et al. 2015; Moseley et al. 2007; Zhao et al. 2008; Whaley 2000]. However, several environments, including embedded systems, may lack the necessary hardware and energy resources.

Offline profiling benefits decline if inputs to the evaluation runs do not match well to those used by the training runs. Other researchers have observed the influence of input arguments on a program's run-time behavior, and explored mechanisms to understand and exploit this influence [Mao and Shen 2009; Tian et al. 2010; Shen et al. 2013; Samadi et al. 2012; Berube 2012]. However, more work is needed to adapt existing input characterization and modeling schemes during offline profiling to several different run-time optimizations and algorithms.

In the future, we plan to: (a) quantify the impact of different amounts of profiling data and/or longer collection periods on the performance of adaptive tasks, (b) determine how to maximize the benefits of profile guidance while balancing the run-time overhead and/or the need for additional hardware resources, (c) devise novel techniques to improve the accuracy of offline profiling strategies so they may be reliably employed irrespective of inputs used during the training and actual program runs, and (d) develop predictive models to overcome the effectiveness issues of incomplete profile data in cases where full and accurate information is too costly or too difficult to collect. Finally, this study relied on traces obtained and/or environment provided by the HotSpot VM and was conducted using solely the DaCapo benchmarks. Future research will generalize our findings for other adaptive tasks, runtimes and programs.

8. CONCLUSIONS

Many performance-critical VM tasks require guidance regarding the *future* program behavior to be most effective. Unfortunately, both offline and online profiling approaches face fundamental limitations in their ability to estimate the desired future behavior. While these limitations are generally known, their implication on the effectiveness of dependent profile-driven VM tasks is not well studied. This work presents a systematic exploration of the impact of the inherent profiling limitations on the performance of three important dependent VM optimizations for DaCapo benchmarks.

This study quantifies the performance impact that intrinsic limitations of profiling techniques have on adaptive tasks. We find that both offline and online profiling can

generate profile data that can enable adaptive tasks to achieve close-to-ideal effectiveness in many cases. However, achieving such results will require the development of new techniques to locate and determine closely representative program inputs for offline profiling, and/or enable comprehensive data collection without incurring prohibitive costs/overhead at run-time for online profiling. Overall, our study concludes that improving the predictability of profile data is important to modern run-time systems, and suggests greater focus on developing and quantifying techniques that can achieve such predictability while minimizing the associated run-time overhead.

REFERENCES

Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. 2015. Page placement strategies for GPUs within heterogeneous memory systems. *SIGPLAN Not.* 50, 4 (March 2015), 607–618.

Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. 1997. Continuous Profiling: Where have all the cycles gone? *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 357–390.

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2000a. Adaptive optimization in the Jalapeno JVM. *ACM SIGPLAN Notices* 35, 10 (2000), 47–65.

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2000b. Adaptive optimization in the Jalapeno JVM: The controller's analytical model. In *3rd ACM Workshop on Feedback Directed and Dynamic Optimization (FDDO '00)*.

Matthew Arnold, Stephen Fink, David Grove, Michael Hind, and Peter F. Sweeney. 2005. A survey of adaptive optimization in virtual machines. *Proc. IEEE* 92, 2 (February 2005), 449–466.

Matthew Arnold and David Grove. 2005. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *Proceedings of the Symposium on Code Generation and Optimization*. 51–62.

Matthew Arnold, Michael Hind, and Barbara G. Ryder. 2002. Online feedback-directed optimization of Java. *SIGPLAN Not.* 37, 11 (2002), 111–129.

Matthew Arnold and Barbara G. Ryder. 2001. A framework for reducing the cost of instrumented code. In *Proceedings of the Conference on Programming Language Design and Implementation*. 168–179.

Paul Berube. 2012. *Methodologies for many-input feedback-directed optimization*. Ph.D. Dissertation. University of Alberta, Edmonton, Alberta, Canada.

Walter Binder. 2006. Portable and accurate sampling profiling for Java. *Softw. Pract. Exper.* 36, 6 (2006), 615–650.

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 Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications (OOPSLA '06)*. ACM, 169–190.

Jacob Brock, Xiaoming Gu, Bin Bao, and Chen Ding. 2013. Pacman: Program-assisted cache management. *SIGPLAN Not.* 48, 11 (June 2013), 39–50.

Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. 1998. Cache-conscious data placement. *SIGPLAN Not.* 33, 11 (Oct. 1998), 139–149.

Bing-Jing Chang, Yuan-Hao Chang, Hung-Sheng Chang, Tei-Wei Kuo, and Hsiang-Pang Li. 2014. A PCM translation layer for integrated memory and storage management. In *Proceedings of the 2014 International Conference on Hardware/Software Codesign and System Synthesis*. ACM, Article 6, 10 pages.

Pohua P. Chang, Scott A. Mahlke, and Wen mei W. Hwu. 1991. Using profile information to assist classic code optimizations. *Software Prac. Experience* 21 (1991), 1301–1321.

Guoyang Chen, Bo Wu, Dong Li, and Xipeng Shen. 2014. PORPLE: An extensible optimizer for portable data placement on GPU. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE, 88–100.

Trishul M. Chilimbi and Ran Shaham. 2006. Cache-conscious coallocation of hot data streams. In *Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 252–262.

Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. 2011. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC '11)*. ACM, 31–40.

Yufei Ding, Mingzhou Zhou, Zhijia Zhao, Sarah Eisenstat, and Xipeng Shen. 2014. Finding the limit: Examining the potential and complexity of compilation scheduling for JIT-based runtime systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, 607–622.

Evelyn Duesterwald and Vasanth Bala. 2000. Software profiling for hot path prediction: Less is more. *SIGPLAN Notices* 35, 11 (Nov. 2000), 202–211.

Evelyn Duesterwald, Calin Cascaval, and Sandhya Dwarkadas. 2003. Characterizing and predicting program behavior and its variability. In *PACT '03: Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*. IEEE, 220–230.

Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically rigorous Java performance evaluation. In *Proceedings of the conference on Object-oriented programming systems and applications*. 57–76.

Darryl Gove and Lawrence Spracklen. 2007. Evaluating the correspondence between training and reference workloads in SPEC CPU2006. *SIGARCH Comput. Archit. News* 35, 1 (March 2007), 122–129.

Rentong Guo, Xiaofei Liao, Hai Jin, Jianhui Yue, and Guang Tan. 2015. NightWatch: Integrating lightweight and transparent cache pollution control into dynamic memory allocation systems. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*. USENIX, Santa Clara, CA, 307–318.

Urs Hoelzle and Luiz Andre Barroso. 2009. *The Datacenter As a Computer: An introduction to the design of warehouse-scale machines* (1st ed.). Morgan and Claypool Publishers.

Peter Hofer and Hanspeter Mössenböck. 2014. Efficient and accurate stack trace sampling in the Java HotSpot virtual machine. In *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE '14)*. 277–280.

Urs Hölzle and David Ungar. 1996. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Trans. Program. Lang. Syst.* 18, 4 (1996), 355–400.

Wei Chung Hsu, Howard Chen, Pen Chung Yew, and Dong-Yuan Chen. 2002. On the predictability of program behavior using different input data sets. In *Proceedings of the Sixth Annual Workshop on Interaction Between Compilers and Computer Architectures (INTERACT '02)*. IEEE, 45–53.

Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. 2004. The garbage collection advantage: Improving program locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '04)*. ACM, 69–80.

Wen-Mei W. Hwu, Scott A. Mahlke, William Y. Chen, Pohua P. Chang, Nancy J. Warter, Roger A. Bringmann, Roland G. Ouellette, Richard E. Hank, Tokuzo Kiyohara, Grant E. Haab, John G. Holm, and Daniel M. Lavery. 1993. The superblock: An effective technique for VLIW and superscalar compilation. *J. Supercomput.* 7, 1-2 (1993), 229–248.

Hiroshi Inoue and Toshio Nakatani. 2009. How a Java VM can get more from a hardware performance monitor. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, 137–154.

Michael R. Jantz and Prasad A. Kulkarni. 2013. Exploring single and multilevel JIT compilation policy for modern machines. *ACM Transactions on Architecture and Code Optimization* 10, 4, Article 22 (Dec. 2013), 29 pages.

Michael R. Jantz, Forrest J. Robinson, Prasad A. Kulkarni, and Kshitij A. Doshi. 2015. Cross-layer memory management for managed language applications. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 488–504.

JEDEC. 2009. DDR3 SDRAM Standard. (2009). http://www.jedec.org/standards-documents/docs/jesd-79-3d

Yunlian Jiang, Eddy Z. Zhang, Kai Tian, Feng Mao, Malcom Gethers, Xipeng Shen, and Yaoqing Gao. 2010. Exploiting statistical correlations for proactive prediction of program behaviors. In *Proceedings of the international symposium on Code generation and optimization (CGO '10)*. 248–256.

Thomas Kotzmann, Christian Wimmer, Hanspeter Mössenböck, Thomas Rodriguez, Kenneth Russell, and David Cox. 2008. Design of the Java HotSpot™client compiler for Java 6. *ACM Trans. Archit. Code Optim.* 5, 1 (2008), 1–32.

Chandra Krintz, David Grove, Vivek Sarkar, and Brad Calder. 2000. Reducing the overhead of dynamic compilation. *Software: Practice and Experience* 31, 8 (December 2000), 717–738.

Prasad A. Kulkarni. 2011. JIT compilation policy for modern machines. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '11)*. ACM, 773–788.

Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. 2003. Energy management for commercial servers. *Computer* 36, 12 (Dec. 2003), 39–48.

Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. 2009. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ACM, 267–278.

Krishna T. Malladi, Benjamin C. Lee, Frank A. Nothaft, Christos Kozyrakis, Karthika Periyathambi, and Mark Horowitz. 2012. Towards energy-proportional datacenter memory with mobile DRAM. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA '12)*. IEEE, 37–48.

Feng Mao and Xipeng Shen. 2009. Cross-input learning and discriminative prediction in evolvable virtual machines. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '09)*. 92–101.

M.R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski, and G.H. Loh. 2015. 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*. 126–136.

Markus Mock, Craig Chambers, and Susan J. Eggers. 2000. Calpa: A tool for automating selective dynamic compilation. In *MICRO 33: Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. 291–302.

Tipp Moseley, Alex Shye, Vijay Janapa Reddi, Dirk Grunwald, and Ramesh Peri. 2007. Shadow profiling: Hiding instrumentation costs with parallelism. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07)*. IEEE, 198–208.

Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the accuracy of Java profilers. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '10)*. ACM, 187–197.

Manjiri A. Namjoshi and Prasad A. Kulkarni. 2010. Novel Online Profiling for Virtual Machines. In *Proceedings of the Conference on Virtual Execution Environments (VEE '10)*. 133–144.

Oracle. 2014. Java virtualmMachine tooliInterface (JVM TI). (December 2014). http://docs.oracle.com/javase/6/docs/technotes/guides/jvmti/.

Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java HotSpot^{TM}server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*. USENIX, 1–12.

Karl Pettis and Robert C. Hansen. 1990. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. 16–27.

Forrest Robinson, Michael Jantz, and Prasad Kulkarni. 2016. Code cache management in managed language VMs to reduce memory consumption for embedded systems. In *to be published in the conference on Languages, compilers, and tools for embedded systems (LCTES '16)*. ACM.

Shai Rubin, Rastislav Bodík, and Trishul Chilimbi. 2002. An efficient profile-analysis framework for data-layout optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '02)*. ACM, 140–153.

Mehrzad Samadi, Amir Hormati, Mojtaba Mehrara, Janghaeng Lee, and Scott Mahlke. 2012. Adaptive input-aware compilation for graphics engines. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '12)*. ACM, 13–22.

Xipeng Shen, Yixun Liu, Eddy Z. Zhang, and Poornima Bhamidipati. 2013. An infrastructure for tackling input-sensitivity of GPU program optimizations. *Int. J. Parallel Program.* 41, 6 (Dec. 2013), 855–869.

Kshitij Sudan, Niladrish Chatterjee, David Nellans, Manu Awasthi, Rajeev Balasubramonian, and Al Davis. 2010. Micro-pages: Increasing DRAM efficiency with locality-aware data placement. *SIGARCH Comput. Archit. News* 38, 1 (March 2010), 219–230.

Kai Tian, Yunlian Jiang, Eddy Z. Zhang, and Xipeng Shen. 2010. An input-centric paradigm for program dynamic optimizations. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '10)*. ACM, 125–139.

Steven Wallace and Kim Hazelwood. 2007. SuperPin: Parallelizing dynamic instrumentation for real-time performance. In *Proceedings of the Symposium on Code Generation and Optimization*. 209–220.

John Whaley. 2000. A portable sampling-based profiler for Java virtual machines. In *Proceedings of the ACM 2000 Conference on Java Grande (JAVA '00)*. 78–87.

Bo Wu, Zhijia Zhao, Xipeng Shen, Yunlian Jiang, Yaoqing Gao, and Raul Silvera. 2012. Exploiting inter-sequence correlations for program behavior prediction. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '12)*. 851–866.

Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2015. Computer performance microscopy with shim. In *Proceedings of the Symposium on Computer Architecture (ISCA '15)*. ACM, 170–184.

Qin Zhao, Ioana Cutcutache, and Weng-Fai Wong. 2008. PiPA: Pipelined profiling and analysis on multi-core systems. In *Proceedings of the Symposium on Code generation and optimization*. 185–194.