Code Cache Management in Managed Language VMs to Reduce Memory Consumption for Embedded Systems

Forrest J. Robinson
EECS, University of Kansas, USA
fjrobinson@ku.edu

Michael R. Jantz
EECS, University of Tennessee, USA
mrjantz@utk.edu

Prasad A. Kulkarni
EECS, University of Kansas, USA
prasadk@ku.edu

Abstract

The compiled native code generated by a just-in-time (JIT) compiler in managed language virtual machines (VM) is placed in a region of memory called the *code cache*. Code cache management (CCM) in a VM is responsible to find and evict methods from the code cache to maintain execution correctness and manage program performance for a given code cache size or memory budget. Effective CCM can also boost program speed by enabling more aggressive JIT compilation, powerful optimizations, and improved hardware instruction cache and I-TLB performance.

Though important, CCM is an overlooked component in VMs. We find that the default CCM policies in Oracle's production-grade HotSpot VM perform poorly even at modest memory pressure. We develop a detailed simulation-based framework to model and evaluate the potential efficiency of many different CCM policies in a controlled and realistic, but VM-independent environment. We make the encouraging discovery that effective CCM policies can sustain high program performance even for very small cache sizes.

Our simulation study provides the rationale and motivation to improve CCM strategies in existing VMs. We implement and study the properties of several CCM policies in HotSpot. We find that in spite of working within the bounds of the HotSpot VM's current CCM sub-system, our best CCM policy implementation in HotSpot improves program performance over the default CCM algorithm by 39%, 41%, 55%, and 50% with code cache sizes that are 90%, 75%, 50%, and 25% of the desired cache size, on average.

Categories and Subject Descriptors D.3 [*Software*]: Programming languages; D.3.4 [*Programming Languages*]: Processors—Compilers, Run-time environments

General Terms Performance, Measurement, Languages

Keywords Code-cache, Memory-constrained, HotSpot JVM

1. Introduction

The rise of Java in the mid-1990's introduced managed runtime environments or virtual machines (VM) to mainstream computing devices, including embedded and mobile systems. High-level managed languages running within a VM, such as Java and JavaScript, have gained extensive adoption since they typically support high-level programming language semantics, portable binary distribution formats, and safe and secure program execution.

VMs execute the portable architecture-independent program binaries using interpretation or binary translation. Program emulation via interpretation is inherently slow [23]. Therefore, modern VMs, like those included with web browsers and most Java virtual machines (JVM), employ just-in-time (JIT) compilation to translate (important sections of) the input binary to native code at runtime [10, 20]. The generated native code is stored in a region of heap memory, called the *code cache*. Thus, the code cache storage enables the native code produced after JIT compilation to be reused later, without re-generating it on every invocation of that region.

JIT compilation consumes computational resources and memory to hold the generated native code at run-time. To trade-off the run-time JIT compilation cost with overall program execution speed, many VMs employ a technique called *selective compilation* to only translate and optimize the frequently used (or *hot*) sections of the program [16, 21]. Unfortunately, even with selective compilation, the memory footprint of the code cache can become significant, especially for memory constrained embedded devices [12, 25]. A large code cache can reduce the memory available to the rest of the executing application, increase the frequency and cost of garbage collection, and decrease overall device response time by lowering the number of programs that are simultaneously resident in memory.

Small embedded devices, such as wearables, often feature powerful multi-core processors, but can only accommodate modest memory capacities. [1] Our measurements reveal that compiling only the hot program methods (with Oracle's production-grade HotSpot *c1* compiler [20]) for just the *startup* run of the standard DaCapo benchmarks [6] results in an average code cache size of over 4MB (see Table 1). Google reported that with Android 4.4, many mobile apps tend to max out the code cache fairly quickly (which by default had been set to 1MB).[2] In fact, with Dalvik, Google recommended the JIT compiler to be entirely disabled for low-memory devices to overcome the increase in memory consumption due to the code cache. However, disabling JIT compilation can significantly degrade program speed. Therefore, it is a critical research challenge to efficiently and accurately determine which methods should reside in the code cache when memory is scarce to maximize overall program performance.

The code cache management (CCM) algorithm was initially designed to maintain program execution correctness in dynamic language VMs by evicting previously compiled regions from the code cache if the assumptions made during compilation are later found

[1] Android smart-watches have adopted dual-core and quad-core ARM Cortex based processors, but typically offer not more than 512MB of memory. https://wtvox.com/smartwatches/best-smartwatch-top-10/

[2] http://source.android.com/devices/tech/config/low-ram.html

to be incorrect. The CCM algorithm in current VMs is also responsible for finding and evicting compiled regions to accommodate native code from later compilations if the code cache is full. The CCM algorithm has a choice when selecting a method to purge from the code cache. Ideally, the algorithm needs to find a method that is not currently hot and will not become hot and trigger a recompilation in the future. Better code cache management can enable the VM to support larger applications, and enhance performance by allowing a greater number of (*phase*-specific) compilations [19], enabling more powerful optimizations (like aggressive inlining for critical methods) [18], and enhancing instruction cache and instruction translation look-aside buffer (I-TLB) performance [12].

In this work we investigate the effectiveness of different CCM strategies to sustain program performance with lower code cache sizes. We find that the default CCM policies supported in the HotSpot JVM produce large performance losses even with modest code cache size pressure. We design a novel simulation-based framework to model and evaluate the potential efficiency of different CCM policies in a controlled and realistic environment that is isolated from VM and hardware specific implementation factors. Encouraging results from this modeling study provide the rationale to design and develop improved CCM methods during actual VM executions. We extend the current CCM algorithm in HotSpot and implement and compare new profiling based CCM policies. Even with minimal changes to the rest of HotSpot's code cache infrastructure, we find that better CCM policies improve average program performance by 39%, 41%, 55%, 58%, and 50% when code cache sizes are limited to 90%, 75%, 50%, 40%, and 25% of the desired cache sizes respectively.

Thus, we make the following contributions in this work:

1. We conduct experiments to measure the impact of constrained code cache sizes on program performance with existing CCM algorithms in HotSpot.
2. We design and build a detailed modeling framework to investigate the effectiveness of different *ideal*, *offline*, and *online-reactive* profiling-based CCM algorithms. The theoretical ideal CCM technique uses knowledge about the *future* program behavior to select the methods to evict, and provides a baseline to compare the efficiency of other practical CCM policies. To understand their potential, our offline and online profiling based CCM models employ the best profiles possible with each technique by discarding the physical costs of profile data collection.
3. We extend existing and implement new CCM policies in HotSpot, evaluate their performance, and assess the impact of profiling overheads and other implementation factors imposed by HotSpot on the effectiveness of CCM techniques.

The rest of this paper is organized as follows. We present background regarding the CCM infrastructure in the HotSpot VM in the next section. We describe the experimental results with current CCM techniques in Section 3. We describe the design of our simulation framework and provide results with the ideal and practical CCM algorithms in Section 4. We explain the HotSpot implementation of CCM policies, and discuss their performance and impact of physical constraints and implementation choices on their effectiveness in Section 5. We present related work in Section 6. Finally, we discuss future work and present our conclusions from this study in Sections 7 and 8 respectively.

2. Background

Our work in this paper employs Oracle's production-grade HotSpot Java virtual machine [20, 22]. In this section we provide a brief background on the internal workings of HotSpot that are relevant to this current work.

Benchmark	Total Methods	Hot methods – startup	
		Num	Size (bytes)
avrora	3,808	630	914,944
fop	7,450	1,573	3,192,320
jython	9,100	2,226	5,574,144
luindex	3,476	532	1,127,936
lusearch	2,901	495	914,944
pmd	5,661	1,758	3,053,056
sunflow	4,457	405	1,133,056
tomcat	13,465	3,092	6,619,948
tradebeans	33,653	3,055	6,008,320
tradesoap	34,319	6,044	12,768,768
xalan	4,815	1,820	3,273,216
Average	10,567	1,741	4,058,717

Table 1. Number of the total and hot program methods, and size occupied by the hot compiled code during the *startup* run for each benchmark.

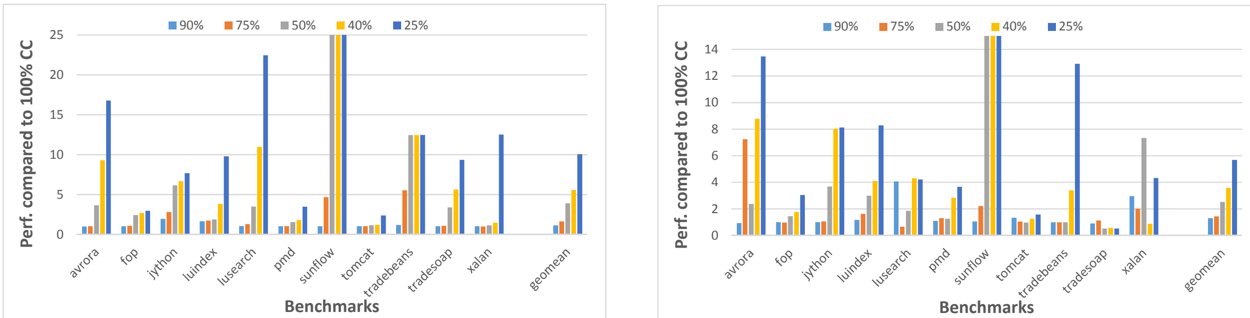
HotSpot's emulation engine includes a high-performance threaded bytecode interpreter and multiple JIT compilers. The execution of a new program begins in the interpreter. HotSpot uses each method's *hotness_count*, which is a sum of the method's *invocation* and loop *back-edge* counters, to promote methods to (higher levels of) JIT compilation. The HotSpot JVM has two dynamic compilers: (a) the *c1* compiler that is quick, and generates code that is lightly optimized and with a smaller memory footprint due to limited inlining, and (b) the *c2* compiler that optimizes code more thoroughly. More recent HotSpot releases support a *tiered* compilation mode that simultaneously enables both compilers to combine their benefits. For this work we only use the c1 compiler to allow easier experimental setup and more precise analysis of observed results. The compilation unit in HotSpot is a single program method.

The compiled code is stored in the code cache. The code cache in VMs typically has a fixed upper bound on size to prevent excessive memory usage. The code cache can contain different code types. For example, HotSpot maintains two primary code types in the code cache: code that is generated by the JIT compilers and persistent infrastructure code generated by the JVM such as adapters and the interpreter. While earlier HotSpot versions had a single unified code cache, the latest HotSpot release implements a *segmented* code cache to segregate the different code types. A segmented code cache has been shown to reduce fragmentation and result in lower I-Cache and I-TLB miss rates [12]. For this work, a segmented code cache makes it easier to precisely control the size of only the segment that holds compiled method code.

In HotSpot, a method selected for eviction by the CCM must transition through several states before actually releasing the memory that it occupies. Each subsequent state transition currently only happens at successive *safepoints*. A CCM algorithm marks a method for eviction by changing its status to *non-entrant*. A non-entrant method cannot be entered, but can exist on the call-stack of an application thread. HotSpot transitions non-entrant methods to the state *zombie* if the method is not on any thread's call-stack. Zombie methods can still be referenced by other methods via inline caches. HotSpot updates the inline cache entries for zombie methods, if any, and then is able to release the space they occupy. Thus, CCM algorithms in HotSpot experience a lag between when method evictions are requested to create free space to when that space actually becomes available to store new compiled code in the code cache.

All our experiments for this work employ 11 DaCapo Java benchmarks with their *default* input size [6].[3] Table 1 shows some

[3] We leave out *batik*, *eclipse*, and *h2* because they fail with the *default* client build of HotSpot-9 without any of our modifications.



(a) (b)

Figure 1. Comparison of benchmark performance (*execution times*) with the existing HotSpot CCM policies, (a) *stop-compiler* and (b) *stack-scan*, different constrained code cache sizes

relevant properties of the different DaCapo benchmarks. For each benchmark in column 1, we show the total number of loaded methods in column 2 of the table. Columns 3 and 4 display the number and size of the compiled (hot) program methods after the *startup* iteration. Many more methods are expected to be compiled by the time the program reaches *steady-state*. All our run-time experiments are performed on a cluster of 8-core 2.84GHz Intel x86-64 machines running Fedora Linux as the operating system. To account for inherent timing variations during the benchmark runs, all the run-time results in this paper report the (geometric) average over 10 runs for each benchmark-configuration pair [9].

3. Current CCM Policies in HotSpot

In this section we assess the effectiveness of existing CCM policies in HotSpot to sustain program performance at different constrained code cache sizes.

We design an experimental setup to systematically limit the code cache size for each program. We first calculate the total accumulated size of all compiled methods in the default *startup* program run for each benchmark, and use it as the full code cache size for that benchmark (100% code cache size). Then, runs with constrained code cache sizes use 90%, 75%, 50%, 40%, and 25% of this full code cache space needed for each benchmark. Thus, the code cache size limits we use are specific to each benchmark.

We evaluate the performance of two CCM strategies, *stop-compiler* and *stack-scan*. The stop-compilation CCM method simply stops all JIT compilation if/when the code cache gets full. This CCM policy is simple and fast, and was therefore employed in several early HotSpot versions and other language VMs, such as Android's Dalvik.

The latest stable HotSpot release uses a profiling-driven adaptive CCM method that we call *stack-scan*. The stack-scan policy uses a separate thread to *sweep* the code cache to remove some of the compiled code when the code cache usage gets close to or over its maximum size limit. The sweeper associates a separate counter with each compiled method in the code cache to keep track of method utilization. This counter is initially set to a high value after method compilation. A method's counter is decremented every time the method is reached during the code cache sweep, and is reset to its original high value if it is found on the call-stack of any application thread. Hot methods are expected to be encountered often on some call-stack and will therefore maintain a high counter value. Methods with lower counter values are candidates from eviction from the code cache when pressure is high. 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.

Figure 1 plots the ratio of run-time program performance with the *stop-compiler* and *stack-scan* CCM strategies at constrained (90%, 75%, 50%, 40%, and 25%) code cache sizes, as compared to a baseline that uses the same (*stop-compiler* or *stack-scan* respectively) CCM policy with 100% code cache size. While the simplistic *stop-compiler* policy can be expected to perform poorly at very low code cache sizes, results in Figure 1 show that both these CCM policies fail to perform satisfactorily even with modest code cache constraints. On average, program performance degrades by 14%, 62.4%, 3.9X, 5.6X, and 10.1X at 90%, 75%, 50%, 40%, and 25% code cache sizes respectively with the simple stop-compiler policy. The more sophisticated *stack-scan* CCM policy does better than *stop-compiler*, but the average program performance still deteriorates by 30%, 43.8%, 2.5X, 3.6X, and 5.7X at our different code cache sizes respectively.

We also observe that even with highly constrained code caches program performance remains significantly better than interpretation alone, showing the importance of JIT compilation. It is interesting to note that with the *stop-compiler* policy, program performance always improves with an increase in code cache size, as expected. However, this property is not maintained by the *stack-scan* policy. To reduce profiling overhead, the *stack-scan* CCM strategy collects and employs imprecise profiling data to guide its eviction decisions. The effect of program performance dropping with an increase in code cache size is a result of imperfect evictions exercised by the *stack-scan* policy due to poor available profile data.

We also notice that unlike the stop-compiler policy that only activates when the code cache gets full, the stack-scan policy is also triggered at high code cache pressures before the cache limit is actually hit. This property of *stack-scan* sweeper results in a slight performance drop even in the 100% code cache case.

Thus, these results reveal that modest and high code cache pressure can have a big negative performance impact with existing CCM strategies. Regrettably, (low cost, but imprecise) program profiling employed by the stack-scan policy appears to not offer acceptable benefit to performance over the stop-compiler method. In the later sections of this paper we explore if more accurate profiling data can enable the VM to more effectively sustain program performance at small code cache sizes.

4. Potential of Profiling Based CCM Policies

Implementation choices can affect the behavior and performance of the CCM sub-system in a VM. For example, the layout of the code cache and the amount of lag between issuing a method eviction request to having space available in the code cache can influence the performance of a CCM policy. Likewise, the cost and proficiency of dynamic profiling at run-time depends on the mechanisms supported in the available hardware and systems software, and are subject to improvement in future systems. It is hard to isolate the effects of such implementation features during actual VM runs to determine the real potential of different CCM strategies to sustain program performance at constrained memory sizes.

Therefore, we build a detailed simulation framework to compare different CCM strategies using offline and online-reactive profiling information in a VM and hardware-independent manner. A simulation framework allows us to effectively control profiling accuracy, cost, and VM implementation factors, while achieving realistic comparisons. Our simulation framework also enables us to design and evaluate the performance impact of an *ideal* profiling strategy that can deliver accurate and timely knowledge of all relevant aspects of *future* program behavior at zero run-time cost. Thus, the ideal CCM policy is able to determine the best methods to evict to minimize the performance impact on future program execution. In this section we describe our simulation framework, discuss the different CCM algorithms that we modeled, and compare their effectiveness to manage program speed at reduced code cache sizes.

4.1 Performance Metric

The simulations need a simple, effective and accurate performance metric to compare the different CCM policies. In this section we describe the performance metric we devise for our simulation runs.

Method Hotness Count: JIT compilation in a (dual-mode) JVM attempts to improve program 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* (method invocation count + loop backedge 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 simulation framework computes the *total program hotness_count* as the measure of the quality of the code cache algorithm.

From Hotness Counts to Execution Time: Ultimately, we are interested in the effect of different CCM policies on program *execution time*. Therefore, we develop a mechanism to associate program hotness_count with execution time.

To relate hotness_counts with program run-time we execute each benchmark with many different configurations and extract the hotness_count and execution time (program wall-time) in each run. Each selected configuration varies some aspect of HotSpot's default CCM algorithm and/or code cache size. We then plot all the points associating hotness_count and run-time for each benchmark, and use the facilities provided by the language 'R' to fit a (quadratic) curve over these points.

Figure 2 shows these plots for (ten) different DaCapo benchmarks (except *tomcat* to allow a nicer fit on the page). The darker band around each curve (too narrow to see on most graphs) plots the 95% *confidence* interval, while the broader lighter band shows the 95% *prediction* interval. Thus, we can see that interpreter hotness_counts are a good indicator of overall program performance, even when the measured execution time includes all aspects of VM execution including JIT compilation, CCM, garbage collection, etc.

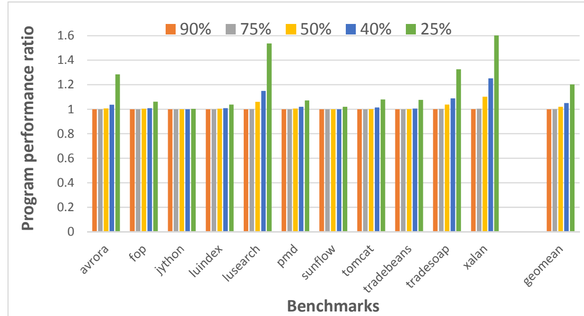


Figure 3. Impact of *ideal* CCM algorithms with different code cache sizes compared to the *ideal* algorithm with unlimited size

The per-benchmark mathematical equation forming the regression curve is used to associate hotness counts with *time* during later simulation runs. We employ this (simulated) time to compare different profiling policies.

4.2 Experimental Setup

In this section we describe the *replay*-based [17] simulation setup we use for our experiments.

Methodology: We instrument HotSpot to generate and log the profile and execution data for our simulation 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*. At the end of each 10msec interval, HotSpot dumps the hotness_counts of all program methods.

The other profile run is to determine the size of the compiled native code for all program methods. We run the HotSpot VM in its default mode, and record the space occupied by the native code generated for each compiled method in the code cache. 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 profile data to simulate the operation of the code cache manager with different method eviction algorithms and different code cache sizes. These runs again use 100%, 90%, 75%, 50%, and 25% of the maximum code cache space needed for each benchmark.

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:

Ideal: This algorithm looks into the *future* profile of the program to find (close to) the ideal 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.

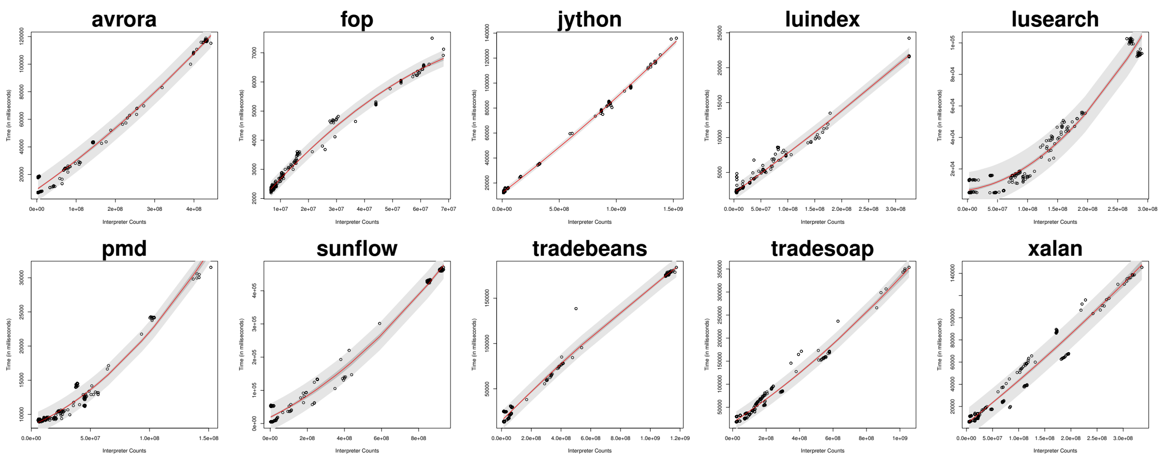


Figure 2. Individual benchmark plots associating hotness counts with program execution time. The X-axis plots the interpreter hotness_counts and the Y-axis shows the corresponding program execution time.

Offline: This set of algorithms attempts to simulate a CCM policy that uses an offline profiling strategy. The 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 a different input for the profiling and measured runs. We use a profile with the DaCapo *small* input for measured runs with 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: These CCM algorithms employ the online reactive profiling strategy, where 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 simple formula finds the hotness_count for each method by assigning progressively lower weights to older profile data:

$$\tau_{n+1} = \alpha * t_n + (1 - \alpha)\tau_n \quad (1)$$

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 several different α values of 0, 0.1, 0.5, 0.9, 1.0. We present the results for $\alpha = 0.1$, which provided the best overall numbers.

Stop compiler: A simple CCM policy that stops JIT compilation when the code cache gets full.

Stack scan: This is an implementation of a simplified version of HotSpot-8's CCM algorithm in the simulator.

4.3 Results and Observations

In this section we present the results of our experiments to evaluate and compare the effectiveness of different CCM algorithms compared with an ideal profiling approach that uses knowledge of the future program behavior.

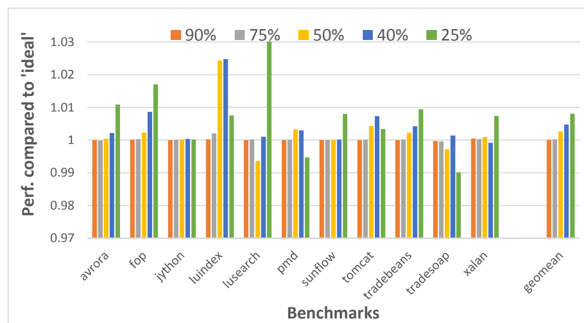


Figure 4. Impact of using the online reactive CCM algorithm compared with the *ideal* algorithm for the same cache sizes

Performance Potential with Ideal CCM Policy Figure 3 shows the potential of *ideal* profiling with CCM at different constrained code cache sizes. Each bar in this graph plots the ratio of the (simulated) program run-time with the ideal CCM policy and indicated code cache size to the run-time with an ideal algorithm and an unlimited code cache. An unlimited code cache never needs to evict compiled methods from the cache. We observe that an ideal 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%, 75%, and 50% of required code cache space. Even with only 40% and 25% of desired code cache size many benchmarks do not see a noticeable performance impact with an (geometric) average performance loss of only 5% and 20% respectively. This result shows that an ideal feedback-driven CCM policy can significantly reduce an executing program's code cache memory requirement with minimal performance losses in most cases.

Performance Potential of Other CCM Policies Next we compare the performance effectiveness of practical CCM policies as compared to the performance delivered by the ideal CCM strategy. The profiling driven CCM algorithms in our simulation framework have access to the most comprehensive, accurate, and timely profile data possible by that profiling technique with no run-time overhead. We have also implemented these policies in HotSpot, and in the next section we present evaluation and analysis of their run-time cost and impact on effectiveness.

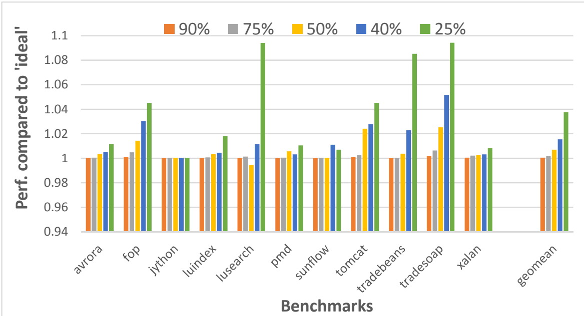


Figure 5. Impact of using the offline-same CCM algorithm compared with the *ideal* algorithm for the same cache sizes

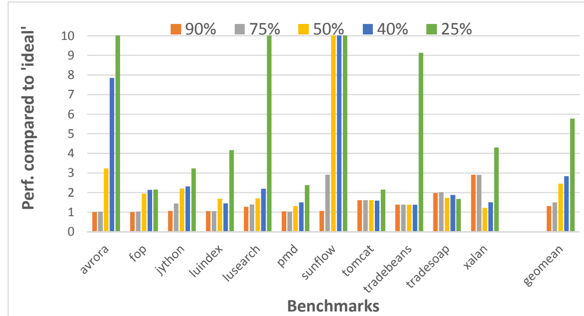


Figure 6. Impact of using our implementation of the HotSpot stack-scan CCM algorithm compared with the *ideal* algorithm for the same cache sizes

Figure 4 shows the performance of the CCM algorithm when using the best *Reactive* profiling strategy (for $\alpha = 0.1$) as *compared with the corresponding ideal approach* for the same code cache sizes. We find that a good reactive strategy can achieve program performance close to ideal even for heavily constrained code cache sizes. The average performance losses compared to ideal with this reactive strategy are only 0.0%, 0.0%, 0.2%, 0.4%, and 0.8% 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 program execution for code cache management. Remember that the cost of collecting this accurate profiling information at run-time is ignored during this simulation study. While we only show the results for the reactive algorithm with an α of 0.1, we note that other reactive schemes also do similarly well.

Figure 5 presents the performance comparison of the *Offline-same* code cache eviction algorithm compared with the corresponding ideal CCM approach. We see that with a perfectly representative offline profile, the CCM algorithm again performs quite well. The *Offline-same* strategy results in an (geometric) average performance loss of 0.0%, 0.1%, 0.6% 1.5%, and 3.8% for our five code cache sizes respectively, compared to the *ideal* algorithm. As opposed to online profiling approaches that collect their data during the same program run, offline profiling strategies require a separate program execution to acquire the desired program behavior data. This data must then be structured and aggregated for used by adaptive VM tasks. This data aggregation can reduce the effectiveness of adaptive tasks by limiting its ability to customize for different sections/phases of the program run. The higher performance loss with the offline profiling based CCM strategy, compared with reactive-CCM, shows this negative impact of profile data aggregation.

Offline profiling suffers from another limitation. A different input set or execution environment can cause the application's runtime behavior to differ from its behavior during the profiling run. The influence of this limitation on the efficiency of the adaptive task will depend on the likeness, of lack thereof, of the profiling and actual evaluation run. We attempt to measure the impact of this limitation with our *offline-diff* CCM configuration when profile data collected during program runs with the *small* DaCapo benchmark inputs are used during evaluation runs with the *default* input. As expected, we see a much more noticeable performance loss with this configuration. We find performance losses of 0.3%, 1.2%, 3.9%, 6.1%, and 10.1%, on average, with the *Offline-diff* scheme compared to ideal for the code cache sizes of 90%, 75%, 50%, 40%, and 25% respectively.

The default *stack-scan* CCM policy in HotSpot uses a low-overhead sampling based profiling mechanism, as explained earlier. The *stack-scan* CCM policy can be considered an instance of a 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 simpler variant of this complex policy in our simulator.

Figure 6 displays the the performance comparison of the *stack-scan* CCM algorithm compared with the corresponding ideal CCM approach. We found that this policy fares quite poorly and achieves performance that is 31%, 50%, 2.44X, 2.83X, and 5.77X worse over the ideal configuration, on average, at 90%, 75%, 50%, 40%, and 25% code cache sizes respectively. Thus, these simulation results do a fair job of tracking the actual HotSpot performance numbers with the stack-scan policy displayed in Figure 1(b).

Additionally, we also simulated the simpler *stop-compiler* strategy that simply stops compilation if the code cache gets full. The *stop-compiler* algorithm is the simplest CCM policy and was found to achieve performance that is 6%, 39%, 3.01X, 3.65X, and 6.93X worse when the code cache is constrained to 90%, 75%, 50%, 40%, and 25% of the needed code cache space respectively, on average.

Other than *stop-compiler*, CCM policies evict program methods when the code cache gets full. These evicted methods will now run in the interpreter. A poor eviction decision (that is, evicting a *hot* method) will result in the method quickly becoming hot again, and will be recompiled. Thus, the greater the number of method evictions and recompilations triggered by a CCM strategy, the poorer is its quality and effectiveness. Additionally, the task of performing method evictions and recompilations will also incur an overhead at run-time, and can be used to further estimate the run-time cost or overhead of each CCM algorithm.

Table 2 shows the average number of methods evicted and recompilations of evicted methods performed by each of our simulated CCM strategies over all benchmark programs. As expected, we find that strategies that result in better performance keep more of the important methods in the cache longer. The *stack-scan* CCM policy is an exception because, unlike the other strategies, it temporarily disables compilation when the code cache gets full. For the remaining CCM algorithms, fewer poor eviction decisions in turn also result in fewer recompilations. We can see that availability of *future* program behavior information allows the *ideal* CCM policy to often evict methods that do not need to be recompiled later, especially at modest memory pressure. The average number of method evictions and recompilations steadily increases with smaller/constrained code cache sizes. In general, more effective CCM strategies predict better eviction candidates, and will likely incur less overhead at run-time and exhibit better overall performance.

In summary, our experiments in this section reveal several interesting and important results.

Strategy	90%		75%		50%		40%		25%	
	Evic	Recom	Evic	Recom	Evic	Recom	Evic	Recom	Evic	Recom
Ideal	48.6	0.9	209.6	8.4	1068.3	477.7	2075.9	1308.5	6301.9	5225.8
Offline-same	149.5	64.5	577.8	299.4	2163.4	1526.8	3903.3	3104.9	10806.5	9695.6
Offline-diff	204.1	117.1	819.9	514.6	2895.1	2218.5	4977.6	4154.2	12969.8	11847.3
React-$\alpha = 0.1$	277.9	105.0	730.6	297.6	2201.8	1413.1	3705.1	2748.6	8729.1	7524.2
Stack-scan	7459.6	6468.4	6957.8	6342.5	4900.9	4398.8	4695.8	4175.9	2594.3	2291.1

Table 2. Average number of method evictions and re-compilations for each code cache management algorithm.

1. We find than an *ideal* CCM strategy with access to detailed profile information regarding future program execution can sustain efficient program performance even with heavily constrained code cache sizes. We expect this observation to fuel much further research in developing practical CCM policies that can realize high program speed and low memory consumption in the code cache in actual VMs.
2. It is encouraging to observe that several profiling-based CCM algorithms can achieve effectiveness close to the ideal policy. However, several hurdles will need resolution to realize these policies in a real VM. Online reactive CCM policies need to overcome the cost of profile collection at run-time. Offline profiling CCM strategies need to not only develop mechanisms to find representative program inputs to generate accurate offline profile data and make it available to the VM at run-time, but may also need to investigate approaches to resolve the profile aggregation effect inherent to offline profiling.
3. Our results also reveal that CCM strategies have the potential to be much more effective than HotSpot's default *stack-scan* CCM policy. The stack-scan policy uses an approximate sampling-based online profiling approach to reduce dynamic overhead. It is unclear if the stack-scan policy's poor performance is due to the imprecise nature of profiling data employed, or if it is caused by implementation decisions in HotSpot. We explore and discuss this issue further in the next section.

5. Performance of Profiling Based CCM Policies in HotSpot

In the last section we evaluated the potential effectiveness of several CCM strategies in a controlled simulation setting that allowed us to ignore profiling costs, and other known and unknown VM implementation issues. These simulation experiments provide encouraging results on the potential of CCM algorithms to sustain acceptable program performance even with very limited code cache sizes. Consequently, we explored and implemented a few CCM policies to understand and assess their behavior in the HotSpot JVM. In this section we present an assessment of an extended *stack-scan*, *reactive*, and *offline* CCM policies implemented in HotSpot.

5.1 Impact of VM Implementation Choices

Design and implementation choices exercised in the VM can make a large impact on the performance delivered by the CCM policies. The method eviction (or *sweeper*) mechanism implemented in the HotSpot JVM differs significantly from the perfect method employed by the simulation algorithm in Section 4. This difference impacts the properties of all CCM policies in HotSpot.

In particular, our simulation algorithm installs the compiled methods at the end of each interval. At that time, if sufficient contiguous space is not available, then the algorithm uses the selected CCM policy to find the methods to evict. These selected methods are then evicted and space to install the new compiled method is created *instantaneously*. The program execution can use the newly compiled method immediately in the next execution interval.

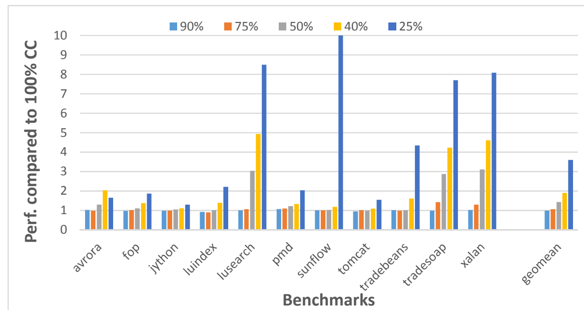


Figure 7. Impact of using the extended SS-no-stop CCM policy at constrained code cache sizes as compared to performance of the same algorithm with maximum (100%) code cache size.

In contrast, HotSpot's sweeper mechanism works differently, and was described earlier in Section 2. With HotSpot, the space needed for future compilations needs to be made available *before* the compiled code is generated. Additionally, method eviction needs to follow several stages from active to non-entrant to zombie, requires several code cache sweeps, and therefore takes some time and is not instantaneous. In our current work, we do not attempt to make changes to the sweeper sub-system in HotSpot. Therefore, all CCM policies implemented in HotSpot have to respect the VM's default sweeper mechanism.

5.2 Stack-Scan No-Stop Compiler (SS-no-stop) CCM Policy

We observe the CCM policy implemented in the latest HotSpot release (called stack-scan) performs poorly with small code cache sizes. These results with the default *stack-scan* policy were presented in Figure 1(b). *Stack-scan* is in fact an instance of a conservative low-cost reactive CCM policy that collects very limited profile information to guide its CCM decisions. Our simulation studies reveal that a reactive CCM strategy (albeit, one with access to detailed profile information) can achieve close to ideal performance numbers. Therefore, it is not entirely clear weather the *stack-scan* policy's lower than expected performance is due to: (a) the quality of employed profile information, or (b) some other implementation factors. We conducted a study to first alleviate the effect of possible implementation factors.

The *stack-scan* CCM algorithm in HotSpot stops the JIT compiler if the code cache gets full. The policy should restart compilation once adequate code cache space becomes available and certain other conditions are satisfied. However, we observed that the compiler restart rarely happens for our benchmarks. We experimented with relaxing the conditions to restart compilation.

Figure 7 plots the performance of our most aggressive extended stack-scan policy that does not stop method compilations even when the cache is full. Generated compiled code that does not find room in the code cache will be discarded. We note that JIT compilation with the c1 compiler is very fast, and we found that the few discarded compilations do not add much overhead to the overall VM execution time. We observe that this simple extension

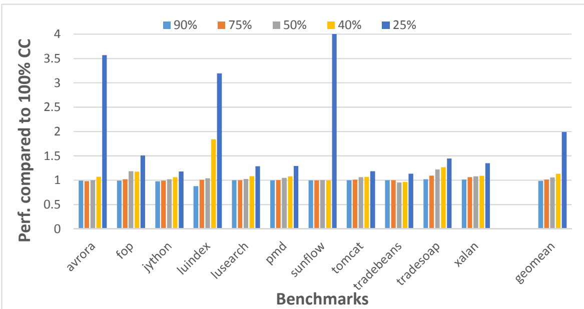


Figure 8. Impact of using our implementation of the *reactive* CCM policy at different constrained code cache sizes as compared to performance of the same algorithm with maximum (100%) code cache size.

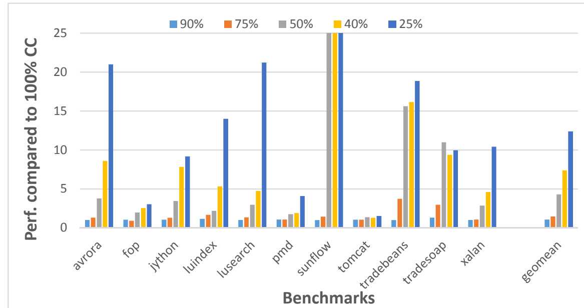


Figure 9. Impact of using our implementation of the *offline-same* CCM policy at different constrained code cache sizes as compared to performance of the same algorithm with maximum (100%) code cache size.

to HotSpot's default stack-scan implementation makes it much more efficient at sustaining program performance at lower code cache sizes. On average, this extended *stack-scan* CCM policy degrades program speed by 0%, 6.1%, 43.1%, 89.9%, and 3.6X when code cache size is restricted to 90%, 75%, 50%, 40%, and 25% respectively, and as compared to a baseline that employs the same CCM policy with 100% code cache size.

5.3 Reactive (Online) CCM Policy

Next, we implement a *reactive* CCM policy in HotSpot based on our *Reactive* simulation setup that collects and employs more comprehensive profile information. This *reactive* CCM strategy implements method-specific counters that are incremented on each method entry and loop back-edge (in both the interpreter and the compiler). We again employ Equation 1 to calculate the *hotness* score of each method on every sweeper activation. This hotness score accounts for the parameter α to appropriately account for the method's recent hotness and past (historical) hotness.

Our modified HotSpot sweeper evicts methods that have the smallest hotness scores until we evict 15% of the code cache (by size), or until the score passes below some costliness threshold. This heuristic allows the policy to keep deleting past 15% of free space as long as those additional methods evicted are cold. We found this heuristic to reduce the number of compile failures, increase responsiveness, and allow the VM to better handle any surges in compilation requests.

Figure 8 plots program performance with the *reactive* CCM policy in HotSpot. Similar to our simulation setup, we again use an α value of 0.1. Thus, we can see that the quality of profile information used by the CCM algorithm has a definite impact on its effectiveness. On average, we find that the *reactive* CCM policy leaves performance unchanged for 90% code cache size, and degrades program speed by 1.5%, 5.5%, 13.6%, and 99.3% with code cache size that is 75%, 50%, 40%, and 25% respectively, when compared to a baseline that employs the same *reactive* CCM policy with 100% code cache size. Note that the selected baseline allows us to ignore the cost of the profiling overhead. Program performance including the profiling overhead is presented and discussed in Section 5.5.

5.4 Offline-Same CCM Policy

An offline profiling based optimization has the benefit that there is no cost of collecting profiling data at run-time, and can simplify VM implementation by removing the need to support any profiling infrastructure in the VM. However, an offline profiling based strategy requires prior training runs, and generally aggregates profile data across the training runs. Profile data aggregation makes it

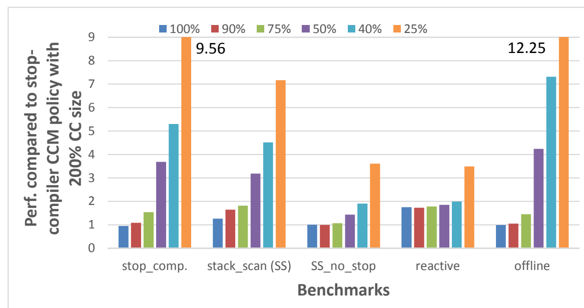


Figure 10. Average impact of the different CCM policies implemented in HotSpot at different constrained code cache sizes as compared to performance of the *stop-compiler* CCM policy with 200% code cache size.

difficult to customize the optimization for different execution-time program phases.

We implemented an *offline* CCM policy in HotSpot based on our *Offline-same* simulation setup. We conduct a single training run in interpretation mode and calculate the overall hotness (invocation + loop back-edge) counts of all program methods. A list of methods sorted in ascending order of their hotness counts is given to HotSpot at the start of the program's evaluation run. The CCM policy evicts methods from the code cache in this provided order.

Figure 9 shows program performance with the *offline* CCM policy in HotSpot. We observe that this strategy does not perform as well as the reactive CCM policies. On average, the *offline* CCM policy drops performance by 5%, 46%, 4.28X, 7.38X, and 12.37X with code cache size that is 90%, 75%, 50%, 40%, and 25% respectively, when compared to a baseline that employs the same *offline* CCM policy with 100% code cache size. Thus, even with perfectly representative offline profile data, our current implementation of this policy in HotSpot fails to deliver acceptable effectiveness. These poor results from the *offline* CCM policy contradict our observations from the simulation studies, and we will attempt to understand and possibly resolve this behavior in future work.

5.5 Overall Comparison of CCM Policies in HotSpot

Figure 10 compares the effectiveness of all the HotSpot CCM policies using a common baseline. The selected baseline is program performance with the simplest *stop-compiler* CCM algorithm and 2X the code cache size desired by each benchmark (200% code cache size). Remember, that 100% code cache size is benchmark-specific, and is computed by summing the sizes of all methods compiled for each benchmark in the default HotSpot configuration. We use 200% code cache size for our baseline because some CCM policies, like *stack-scan*, activate when available free cache space approaches some threshold of allocated cache size, and therefore trigger even with the 100% code cache size configuration.

From Figure 10 we can observe that the *stop-compiler* and *offline* CCM policies only achieve acceptable performance at very modest memory pressure, when most methods are able to reside in the cache. At higher memory pressures, these policies degrade quickly and significantly. The comprehensive profile data available to the *reactive* policy allows it to make excellent decisions about which methods to evict, but the overhead of incrementing counters at every method entry and loop back-edge hurt execution time. Only at very heavily constrained code cache sizes does the benefit of better eviction decisions overcome the profiling cost with this strategy. In future work, we will further investigate the trade-offs between profile quality and cost for reactive CCM policies. Finally, HotSpot's *stack-scan* is an implementation of a low-cost reactive CCM strategy that collects and uses approximate profile data. The effectiveness of HotSpot's default *stack-scan* policy improves significantly with our extensions, and this *SS-no-stop* policy achieves the best or close-to-best overall performance results for most code cache sizes. Compared to the default HotSpot policy, the *SS-no-stop* CCM implementation improves performance by 20.4%, 39.0%, 41.3%, 54.9%, 57.8%, and 49.7% at our various code cache pressures respectively, on average.

It is important to appreciate that all these policies still perform much better than completely disabling JIT compilation and only using the interpreter. On average across all our benchmarks, interpreter-only execution time is 17.35 times worse than the *stop-compiler* CCM policy with 200% code cache size.

6. Related Work

Code caches are used to store translated and/or optimized code in managed language VMs and dynamic binary translators (DBT). Researchers in both these related areas have previously investigated issues regarding code cache layout and CCM to reduce memory consumption. In this section we present and compare past research that is related to our current work in this paper.

Zhang and Krintz were among the first researchers to study and present the effect of method eviction from the code cache on memory consumption and program speed in a JVM [25, 26]. Similar to our present research, this work evaluated the efficiency of offline and online profiling techniques to find the appropriate set of methods to evict from the code cache. Additionally, they also studied techniques to decide when to invoke their eviction algorithm. However, this work was conducted in a compile-only JVM (Jikes RVM [1, 2]), which presents many different properties compared to the HotSpot JVM that employs a baseline interpreter and only compiles the hot program methods. The influence of a compile-only JVM, and Jikes in particular, cause critical differences in the profiling techniques employed and experimental setup used as compared to our current research. Moreover, the simulation studies are another unique contribution of our work that investigate the potential and properties of many different CCM algorithms in a controlled and VM-independent environment.

Several researchers have explored code cache eviction techniques for DBTs. Hazelwood and Smith found that a medium-grained FIFO eviction scheme achieved better performance than a single-block FIFO scheme by lowering replacement overhead [13]. Dynamo conducts a full cache flush at anticipated program phase changes when the trace generation rate becomes high [5]. Intel's Pin DBT also supports a full code cache flush [15]. DynamoRio adaptively scales up the code cache size based on the program's working set size, but does not implement algorithms to evict compiled blocks to reduce memory consumption in the code cache [7]. The Strata DBT implements techniques to bound code cache memory usage by reducing the space required for DBT-injected code [4]. Hazelwood and Smith proposed a generational code cache that can transition methods from a nursery cache to a persistent cache and evict unused code blocks from the cache [14]. Guha et al. designed a least-recently-used (LRU) profiling policy to selectively (or partially) flush code cache blocks for their DBT [11]. However, DBT code caches store blocks or traces instead of program methods, have fine-grained inter-block linking, and, in general, have different requirements compared to a managed-language JVM.

The organization of the code cache can influence the feasibility and effectiveness of CCM algorithms. Jikes RVM allocates compiled native code to Java objects that are then placed on the common heap with other data objects [1]. Jikes can then use the garbage collector (GC) to *manage* code cache objects and evict unused compiled code. Thus, low memory consumption by code cache objects can enable the Jikes RVM to place more data objects or reduce the frequency of GC [25]. While HotSpot earlier employed a single unmanaged code cache, the latest HotSpot release now employs a segmented code cache, with each segment servicing a distinct type to code [12]. Oracle's Maxine JVM also partitions their code cache into different regions for holding the VM's code, and that generated by its two compilers [24]. Most DBTs employ a single code cache, but may use either a simpler thread-private or a more space efficient thread-shared configuration [8]. The Strata DBT introduced a code cache organization split between the scratchpad and main memory to mitigate performance overhead on embedded systems [3]. We do not vary the default code cache organization in our current work, but plan to explore more effective code cache designs in the future.

7. Future Work

There are many avenues for future work on this topic. Our immediate plan is to further study the properties and improve the implementation of CCM policies in HotSpot, and add other policies, such as FIFO. Second, there is little current research to dynamically find the optimal code cache size for individual program executions in a VM. We will investigate techniques to adaptively and quickly find the ideal balance between performance and code cache size for each program at run-time for memory sensitive embedded devices. Third, a smaller code cache can result in more method evictions and recompilations. Our current work did not measure the effect of a smaller cache size on energy consumption with different CCM policies, which we plan to do in the future. Fourth, the placement of native code in a code cache can influence the amount of cache fragmentation and achieved I-Cache and I-TLB performance. We plan to better understand the impact of these tradeoffs and develop new JIT compilation orders or native code placement techniques in the code cache to optimize these performance factors. Fifth, we will study mechanisms to derive accurate and low-cost profiling data, and explore issues such as the tradeoff between profile data accuracy, quality and performance benefit. Finally, the code cache subsystem includes many components, including the CCM algorithm to find methods to evict, method eviction strategy, and code cache layout. In the future we plan to study and redesign all these components together to find the best overall strategy.

8. Conclusions

The goal of this work is to understand the potential and evaluate the effectiveness of different CCM policies to sustain program performance when code cache sizes are too constrained to hold all the desired hot methods during program execution in a managed-language VM. We design a creative simulation setup to investigate the potential of an *ideal* and many other *practical* CCM policies.

We discover that an ideal CCM strategy can allow the VM to maintain close-to-full program speed even with high code cache memory pressure. Furthermore, we found that profiling-based practical CCM policies can realize close to ideal results.

Unfortunately, the current CCM strategy in the popular HotSpot Java VM, based on a low-cost approximate reactive profiling mechanism, produces large program slow-downs at small code cache sizes. We investigate this disparity in HotSpot's CCM strategy. We implement extensions to HotSpot's default CCM policy and design and re-engineer our other simulated CCM policies in HotSpot. Our CCM algorithms in HotSpot deliver positive results and uncover many other interesting questions that will need resolution to find an optimal CCM strategy for future runtime systems.

The abundance of managed languages and the expectation from small embedded devices to simultaneously support multiple resource-consuming programs makes memory capacity management an important issue for embedded systems. We hope that our work can guide researchers to develop/provide the necessary hardware and software structures to maximize the efficiency of CCM techniques for memory constrained embedded systems.

Acknowledgments

We thank the anonymous reviewers for their thoughtful and constructive feedback. This research is supported in part by the National Science Foundation under CAREER award CNS-0953268, and award CNS-1464288.

References

[1] B. Alpern, C. R. Attanasio, A. Cocchi, D. Lieber, S. Smith, T. Ngo, J. J. Barton, S. F. Hummel, J. C. Sheperd, and M. Mergen. Implementing JalapeÑo in Java. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, pages 314–324, 1999.

[2] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney. Adaptive optimization in the JalapeÑo JVM. In *Proceedings of the 15th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '00, pages 47–65, 2000.

[3] J. A. Baiocchi and B. R. Childers. Heterogeneous code cache: Using scratchpad and main memory in dynamic binary translators. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 744–749, 2009.

[4] J. A. Baiocchi, B. R. Childers, J. W. Davidson, and J. D. Hiser. Reducing pressure in bounded DBT code caches. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 109–118, 2008.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: A transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 1–12, 2000.

[6] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. 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, pages 169–190, 2006.

[7] D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '03, pages 265–275, 2003.

[8] D. Bruening, V. Kiriansky, T. Garnett, and S. Banerji. Thread-shared software code caches. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '06, pages 28–38, 2006.

[9] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 57–76, 2007.

[10] Google. Chrome V8 JavaScript VM, September 2012. https://developers.google.com/v8/intro.

[11] A. Guha, K. Hazelwood, and M. Soffa. Balancing memory and performance through selective flushing of software code caches. In *Proceedings of the 2010 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '10, pages 1–10, 2010.

[12] T. Hartmann, A. Noll, and T. Gross. Efficient code management for dynamic multi-tiered compilation systems. In *Proceedings of the 2014 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '14, pages 51–62, 2014.

[13] K. Hazelwood and J. E. Smith. Exploring code cache eviction granularities in dynamic optimization systems. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 89–99, 2004.

[14] K. Hazelwood and M. D. Smith. Managing bounded code caches in dynamic binary optimization systems. *ACM Transactions on Architecture and Code Optimization*, 3(3):263–294, Sept. 2006. ISSN 1544-3566.

[15] K. Hazelwood, G. Lueck, and R. Cohn. Scalable support for multithreaded applications on dynamic binary instrumentation systems. In *Proceedings of the 2009 International Symposium on Memory Management*, ISMM '09, pages 20–29, 2009.

[16] U. Hölzle and D. Ungar. Reconciling responsiveness with performance in pure object-oriented languages. *ACM Transactions on Programming Language Systems*, 18(4):355–400, 1996. ISSN 0164-0925.

[17] X. Huang, S. M. Blackburn, K. S. McKinley, J. E. B. Moss, Z. Wang, and P. Cheng. 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, pages 69–80, 2004.

[18] H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A trace-based Java JIT compiler retrofitted from a method-based compiler. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, pages 246–256, 2011.

[19] M. R. Jantz and P. A. Kulkarni. Exploring single and multilevel JIT compilation policy for modern machines. *ACM Transactions on Architecture and Code Optimization*, 10(4):22:1–22:29, Dec. 2013.

[20] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1), 2008.

[21] C. Krintz, D. Grove, V. Sarkar, and B. Calder. Reducing the overhead of dynamic compilation. *Software: Practice and Experience*, 31(8): 717–738, December 2000.

[22] M. Paleczny, C. Vick, and C. Click. The Java HotSpot™server compiler. In *JVM'01: Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium*, pages 1–12, Berkeley, CA, USA, 2001.

[23] J. Smith and R. Nair. *Virtual Machines: Versatile Platforms for Systems and Processes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005. ISBN 1558609105.

[24] C. Wimmer, M. Haupt, M. L. Van De Vanter, M. Jordan, L. Daynès, and D. Simon. Maxine: An approachable virtual machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization*, 9(4): 30:1–30:24, Jan. 2013. ISSN 1544-3566.

[25] L. Zhang and C. Krintz. Adaptive code unloading for resource-constrained JVMs. In *Proceedings of the 2004 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES '04, pages 155–164, 2004.

[26] L. Zhang and C. Krintz. Profile-driven code unloading for resource-constrained JVMs. In *Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java*, PPPJ '04, pages 83–90, 2004.