Efficient Code Caching for an Embedded Dynamic Adaptive Compiler

Oleg Pliss, Bernd Mathiske, *Sun Microsystems, Inc.*

Abstract

We report on the code caching techniques in an embedded Java^{TM}Virtual Machine with a dynamic adaptive compiler that is in use in memory-constrained devices such as mobile phones. In this setting, compiled code cache management can be performance-critical.

We introduce a combination of sampling and instrumentation techniques with very low mutator overhead to profile frequently and recently executed methods. Building on this, we present an eviction policy featuring a weighting and decay system that accounts for both recency and frequency of method execution.

Combining the above with garbage collector feedback, the compiled code cache can be dynamically adjusted in size while competing with the object allocator for memory.

Due to our improvements in accurate working set detection and cache size management, our JVM maintains high execution speeds at significantly reduced heap sizes and the eventual performance degradation is smoother.

Running all EEMBC benchmarks for CLDC successively, we observed large performance increases compared to previous versions of the same JVM, especially with small heap sizes below 500 KB and with very large ones above 2 MB.

1 Introduction and Motivation

The first generation of Java^{TM}-enabled wireless devices used variants of the KVM, which features an interpreter loop, but no JIT. Using dynamic adaptive compilation, the CLDC^{TM}HotSpot Implementation delivers an order of magnitude increased performance compared to the KVM while maintaining the constraining footprint requirements of small devices such as mobile phones [Sun03].

A major implementation challenge when building this high-performance JVM was the creation of a compiled code caching scheme that would sustain high performance levels under varying memory availability.

Given a dynamic adaptive compiler, the working set of methods in execution regularily consists of both interpreted and compiled methods. Any interpreted method is compiled when it is found to be executing when a periodic sampling occurs or when its execution counter exceeds a threshold.

Aiming to increase overall performance by minimizing the portion of time spent interpreting byte codes, we are focussing in this paper on working set detection and cache eviction policy.

We regard as *working set* the set of methods of which any code has been executed within an interval of a certain length. This interval is selected to be in the immediate past and its length needs to be set so that the determined working set predicts the next working set as closely as possible.

2 Profiling

A prototype version of our JVM used to sample method execution over the entire interval between full garbage collections by our generational collector. We observed that the sheer length of these intervals typically rendered most of the gathered profiling information useless, i.e. there was insufficient differentiation of recency to support informed eviction decisions. Furthermore, frequency was only accounted for in terms of executing during *several* full collections.

We know that many other VMs avoid the above problems by instrumenting generated code (and the interpreter) to record execution events and decaying the latter over time. Whereas this approach is capable of gathering sufficiently accurate information, compiled code instrumentation causes noticable performance reduction.

Another alternative is sampling, i.e. inspecting the stack contents in periodic intervals. Apparently less intrusive than instrumentation, this approach may suffer from either scarcity of information when the sampling rate is too low or unacceptable overhead when the sampling rate is too high. We refrained from attempting to tune this technique, because it appears to be unstable with varying applications.

Our solution in our current production JVM consists in a combination of some of the above techniques. First, we chose interpreter instrumentation, which has benign overhead since by far most execution happens in compiled code. Interpreted methods are recorded in a short, cyclical buffer. Compiled code is also instrumented, but with the shortest possible insertion of code, a single instruction that updates a flag in a global array, which represents activity sensor flags for all cache-resident compiled methods.

In principle, instrumentation could be placed at certain subsets of 3 out of the following 5 situations to achieve complete coverage of possible program execution: calling a method, getting called, returning from a method, getting returned to, backwards branch. Our experiments indicate that picking just one of them, instrumenting the method prologue, yields practically the same quality of results as any other combination.

Avoiding the pitfalls of long observation intervals, we summarize the gathered activity flags at periodic, relatively short intervals. Thus we combine instrumentation and sampling.

The concrete interval length is mainly device-dependent and is tuned once per installation on a different hardware device. Typical lengths on mobile phones are in the low tens of milliseconds.

3 Compiled Method Weighting

According to [ST85], LRU eviction comes close to the theorectical optimum and frequency is less important than recency. Furthermore, on phase transitions, frequency can be even more misleading than recency. Frequency is nevertheless rather important in our case, because the performance payoff of compiling hot methods can be enormous. (In contrast, paging mechanisms in OSs can ignore frequency, as they must cache any given page even on a single access.)

As an exact implementation of LRU would be too expensive, we use an approximation based on a fixed point number weighting system for compiled methods.

The integer part of each weight reflects the frequency of very recent activity. The fractional part stores a measure of activity further in the past. Currently we use 2 integer bits and 4 fractional bits. Weights are updated on every sampling event and on every attempt to compile a method, which together represent cache time progression.

During a sampling event, we add 1.0 to the weight of every method that has its flag set in the activity sensor array. If any weight overflows, all weights are normalized by a logical shift right by one bit. The primary reason for decaying weights is the limited weight precision. This can be interpreted as when the amount of recent activity exceeds the limit, time has to move forward and so some activity is shifted to the past.

Though frequency of weight updates is driven by compiler activity, this does not yet take into account the interpreted portion of the working set.

Decay progression is supplemented by the interpreter: if the number of execution events in its log exceeds a certain limit, decay is triggered. This feature makes the cache more responsive to new compilations and ensures quick phase transitions, in which the interpreted portion of the working set quickly grows temporarily. It also deals with the rare pathological case when uncompilable (e.g. extremely large) methods keep preventing compiled code execution and thus would effectively stall cache time.

Weights are organized as a linear fixed-size byte array. During profiling summarization and decay processing, weights are accessed efficiently by quads (double words).

4 Method Eviction

On small devices, only a fairly limited amount of memory is available for compiled code caching. Additionally, we need to keep the code cache small to relieve pressure on the garbage collector, since in our JVM, we allocate all compiled methods on the object heap. This uniform resource management not only supports scaling down to rather small devices, it also provides a handy way to adjust the size of the code cache.

We base the size target for the code cache on a heap occupancy prediction, which is an extrapolation of the observed occupancy after the most recent GC. Thus we avoid instrumenting the marking phase of the GC to determine the actual current occupancy.

During every GC, we try to "right-size" the code cache: large enough to capture the current working set, but small enough to avoid depleting heap memory available to the mutator. In this scenario, method eviction can be seen as a side effect of cache size adjustment after the mutator had time to grow the cache (and prepare victim selection).

To evict a specified amount of memory we logically sort methods by their weights and evict methods one by one starting from the lowest weight until the requested amount is freed or there are no more methods left except currently executing ones. Instead of actually sorting, we quickly build and scan a histogram of allocated method space per weight.

Currently executing methods are retained in the cache by setting the high bit of their weight, so that they automatically supercede any other methods.

Finally, we need to address potential eviction thrashing caused by a working set that is larger than our target cache size. If we cannot increase the latter, we let compilation and cache insertion only proceed if there is a prospect of freeing enough space due to a sufficient amount of cache-resident methods with relatively low weights.

5 Related Work

[CCK+03] present several alternative eviction strategies, including an LRU-like weighting system for interpreted and compiled methods, which are kept in different memory areas.

There are many related efforts in operating systems concerning virtual memory. See for instance [Kap99] for a comprehensive classification of eviction policies.

References

[CCK+03] G. Chen, G. Chen, M. Kandemir, M. Vijaykrishnan, and M.J. Irwin. Energy-Aware Code Cache Management for Memory-Constrained Java Devices. In *Proceedings of the IEEE International SOC Conference (ASIC/SOC'03), Portland, Oregon*, September 2003.

[Kap99] S.F. Kaplan. *Compressed Caching and Modern Virtual Memory Simulation.* PhD thesis, The University of Texas at Austin, Texas, 1999.

[ST85] D.D. Sleator and R.E. Tarjan. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, 28(2):202–208, February 1985.

[Sun03] The CLDC HotspotTM Implementation Virtual Machine. Technical report, Sun Microsystems, Inc., 2003. White Paper.