Stop and Go: Understanding Yieldpoint Behavior

Yi Lin[†] Kunshan Wang[†] Stephen M. Blackburn[†] Antony L. Hosking[*] Michael Norrish[‡]

[†]Australian National University [*]Purdue University, USA [‡]NICTA, Australia

[†]{yi.lin,kunshan.wang,steve.blackburn}@anu.edu.au [*]hosking@purdue.edu

[‡]michael.norrish@nicta.com.au

Abstract

Yieldpoints are critical to the implementation of high performance garbage collected languages, yet the design space is not well understood. Yieldpoints allow a running program to be interrupted at well-defined points in its execution, facilitating exact garbage collection, biased locking, on-stack replacement, profiling, and other important virtual machine behaviors. In this paper we identify and evaluate yieldpoint design choices, including previously undocumented designs and optimizations. One of the designs we identify opens new opportunities for very low overhead profiling. We measure the frequency with which yieldpoints are executed and establish a methodology for evaluating the common case execution time overhead. We also measure the median and worst case time-to-yield. We find that Java benchmarks execute about 100 M yieldpoints per second, of which about $1/20\,000$ are taken. The average execution time overhead for untaken yieldpoints on the VM we use ranges from 2.5 % to close to zero on modern hardware, depending on the design, and we find that the designs trade off total overhead with worst case time-to-yield. This analysis gives new insight into a critical but overlooked aspect of garbage collector implementation, and identifies a new optimization and new opportunities for very low overhead profiling.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection), Run-time environments

General Terms Experimentation, Languages, Performance, Measurement

Keywords yieldpoints, safe points, code patching, managed code, managed run-time

ISMM'15, June 14, 2015, Portland, OR, USA
ACM. 978-1-4503-3589-8/15/06
http://dx.doi.org/10.1145/2754169.2754187

1. Introduction

A *yieldpoint* is a frequently executed check by managed application code in high performance managed run-time systems, used to determine when a thread must yield. Reasons to yield include exact garbage collection, user-level thread preemption, on-stack replacement of unoptimized code with optimized code, biased locking, and profiling for feedback directed optimization. Yieldpoints ensure that each thread is in a state that is coherent for the purposes of the yield, such as knowing the precise location of all references in the registers and stacks for exact garbage collection, and that relevant operations such as write barriers and allocation have completed (*i.e.*, are not in some inconsistent partial state). These properties are less easily assured if threads suspend at arbitrary points in their execution. Coherence is essential when the virtual machine needs to introspect the application thread or reason about interactions between the thread and the virtual machine or among multiple application threads. In the case of exact garbage collection, yieldpoints are known as *GC-safe points* [11]. Compilers generate a GC *map* for each yieldpoint, allowing the run-time system to identify heap pointers precisely within the stacks and registers of a yielded thread.

To avoid unbounded waits, yieldpoints typically occur on loop back edges and on method prologs or epilogs of the application, either in the interpreter or in code placed there by the compiler. Consequently, yieldpoints are prolific throughout managed code. Yieldpoints may also be performed explicitly at other points during execution, such as at transitions between managed and unmanaged code.

Despite their important role, to our knowledge there has been no detailed analysis of the design space for yieldpoints nor analysis of their performance. This paper examines both.

We conduct a thorough evaluation of yieldpoints, exploring how they are used, their design space, and performance. We include designs that to our knowledge have not been evaluated before, as well as two designs that are well known. We start by measuring the static and dynamic properties of yieldpoints across a suite of Java benchmarks. We measure their static effect on code size as well as the dynamic rate at which yieldpoints are executed and the rate at which a yieldpoint's slow path is taken (making the thread yield). Statically, yieldpoints account for about 5 % of in-

structions. The Java benchmarks we evaluate perform about 100 M yieldpoints per second, of which about $1/20000$ are taken. We show that, in our system, among the different uses of yieldpoints, by far the most common reason for yielding is to perform profiling for feedback directed optimization (FDO), which in our Java run-time system occurs once every 4 ms. By comparison, garbage collection occurs far less frequently, and in most benchmarks lock revocation in support of biased locking is very rare.

We examine the design space, including two major dimensions. The first dimension is the mechanism for deciding whether to yield, which may be implemented as: (*i*) a conditional guarded by a state variable, (*ii*) as an unconditional load or store from/to a guard page, or (*iii*) via code patching. The conditional yields when the state variable is set and a branch is taken, the unconditional load or store yields when the guard page is protected and the thread is forced to handle the resulting exception, while code patching can implement a branch or a trap (both unconditional). The second design dimension is the scope of the signal, which may be global or per-thread. A global yieldpoint applies to all threads (or none), while a per-thread yieldpoint can target individual threads to yield.

We identify a new opportunity for yieldpoint optimization. Rather than using code patching to turn unconditional yields on or off (which requires that *all* yieldpoints be patched) as Agesen [1] did, we can use code patching to selectively replace frequently executed yieldpoints with noops. We also show that a yieldpoint implemented as an unconditional store can serve double-duty as a very low overhead profiling mechanism. If the unconditional store writes a constant that identifies characteristics of the particular yieldpoint (*e.g.*, location or yielding thread), then a separate profiling thread can sample the stores and thus observe the yieldpoints as they are traversed. We make use of this yieldpoint in separately published work [19].

We evaluate each of the design points and explore the potential for code patching as an optimization. Among these designs, the most important tradeoff is due to the choice of mechanism, with explicit checks incurring the highest overhead in the common untaken case, around 2 %, but delivering the fastest time-to-yield, while the unconditional load or store has a lower overhead in the common case, 1.2 % at best, but has worse time-to-yield performance. The code patching yieldpoint is slightly different than the other yieldpoint designs. Code patching yieldpoints have superior common case overhead when implemented as noops, but the cost of patching *all* yieldpoints outweighs any benefit on mondern hardware. We also evaluate the tradeoffs inherent to using code patching as an optimization.

Our analysis gives new insight into a critical but overlooked aspect of garbage collector implementation, identifies a new yieldpoint optimization, and new opportunities for very low overhead profiling.

2. Background, Analysis, and Related Work

We now describe yieldpoints in more detail and quantitatively evaluate how yieldpoints are used in Java workloads.

2.1 Background

In principle, language run-times are concurrent. This is clear in the case of languages such as Java that support concurrency, but even in the case where the supported language offers no application-level concurrency, such as JavaScript, the relationship between the application code and the underlying run-time system is fundamentally concurrent. The concurrency may be explicit, with run-time services executing in discrete threads or it may be implied, with the underlying run-time services and the application interleaving their execution by time-slicing a single thread. Yieldpoints are a critical mechanism for coordinating among application threads and the run-time system.

Yieldpoints serve two complementary goals. First, they provide precise code points at which the execution state of each application thread is observably coherent, allowing the possibility of unobserved incoherent states between yieldpoints. For example, by ensuring that garbage collection only occurs at yieldpoints, we are assured that a multi-instruction write barrier will be observed in its entirety or not at all. Second, yieldpoints reduce the cost of maintaining metadata with which the thread's state may be introspected. In general, introspection of an application thread depends on metadata (*e.g.*, stack maps) to give meaning to the machine state of the application at any point in time. For example, the type of a value held by a machine register at a given moment will determine whether the value should be interpreted as a pointer, in which case its referent must be retained by the garbage collector, or a floating point number, in which case the value must not be altered. Because such metadata is expensive both in terms of space and in the engineering overhead of coherently generating and maintaining it, language run-times typically only maintain such metadata for a limited set of code locations.

When yieldpoints are used to coordinate garbage collection it is typically adequate for the yield to have global scope — when activated, all application threads yield to the collector. However, when yieldpoints are used for one-to-one interactions between threads, such as for lock revocation in support of biased locking [14], or to support work-stealing [12], a per-thread scope is necessary for good performance. These considerations affect the yieldpoint design space, which is discussed in Section 3.

Yieldpoints are either *injected* into the application execution by the interpreter or compiler, or they are *explicit*, called by the underlying run-time at key points such as transitions into and out of native code. The focus of our study is *injected* yieldpoints, which are prolific.

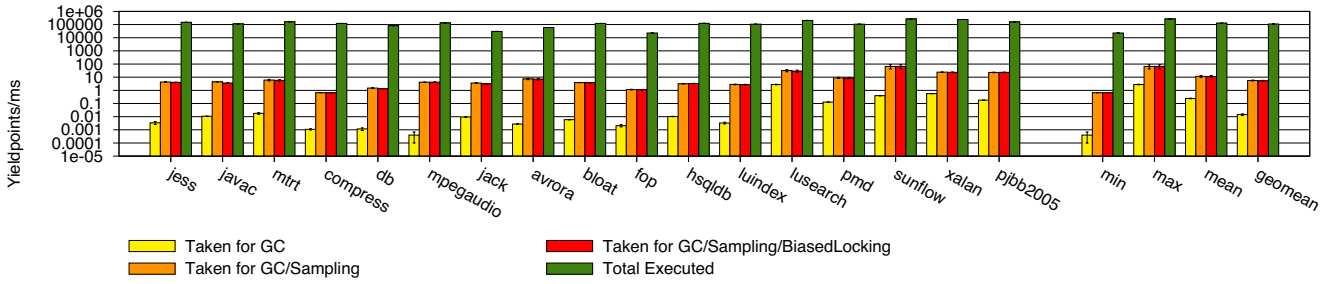


Figure 1. Dynamic yieldpoint rates per millisecond for Java benchmarks, showing those taken due to GC (yellow), those taken due to GC *or* sampling for FDO (orange), all taken yieldpoints (red), and all yieldpoint executions, whether taken or not (green). Counts are per thread, so multi-threaded benchmarks such as lusearch and xalan show noticeably higher take rates, reflecting their higher thread count.

2.2 Analysis

We now present an analysis of the prevalence of yieldpoints, dynamically and statically, and the rate at which yieldpoints are taken. We use a suite of Java benchmarks and instrument a virtual machine to count yieldpoints. Because the instrumentation slows the virtual machine significantly, we use execution times for the uninstrumented virtual machine as our baseline when measuring rates. The details of our methodology are presented in Section 4.

To measure the static impact of yieldpoints on code size, we compiled a large body of Java code using our virtual machine's optimizing compiler (with `O2` optimization level) and found that the resulting machine code increased in size from 13.6 MB to 14.6 MB (*i.e.*, by 7.2 %) when a basic conditional yieldpoint was injected on each loop back edge, method prolog and epilog.

We measure the dynamic impact of yieldpoints by instrumenting the injected code to count the number of times injected yieldpoints are executed, and the number of times yieldpoints are taken for FDO profiling, lock revocation, and garbage collection. We used the execution time for uninstrumented code to determine yieldpoint rates.

Figure 1 shows the rate at which yieldpoints are executed and taken across the suite of Java benchmarks per millisecond. The green bars indicate the rate at which yieldpoints are executed, taken or not. On average about 100 M yieldpoints are executed per second; about one every 10 ns, which is roughly one every 40 cycles on our 3.4 GHz machine. Of these, around 1/20 000 yieldpoints are taken. Sampling for FDO (orange and red bars) dominates the reasons for yieldpoints to be taken. Our virtual machine uses bursty sampling [4], initiating sampling on each thread once every 4 ms. Once initiated, samples are taken at the next N method prologs, where N is 8. The degree of simplicity and longevity of the benchmarks affects the precise number of samples taken. Our counts are totals across all threads, so the multi-threaded benchmarks such as lusearch, sunflow, xalan and pjbb2005 have their counts inflated in proportion to the number of threads they are running. The yellow bar indicates the number of yieldpoints due to garbage collection, and reveals that only a small fraction of taken yieldpoints are due to garbage collection. The difference between red and orange bars reflects the number of yieldpoints taken due to lock revocation, revealing that this is very rare among our benchmarks.

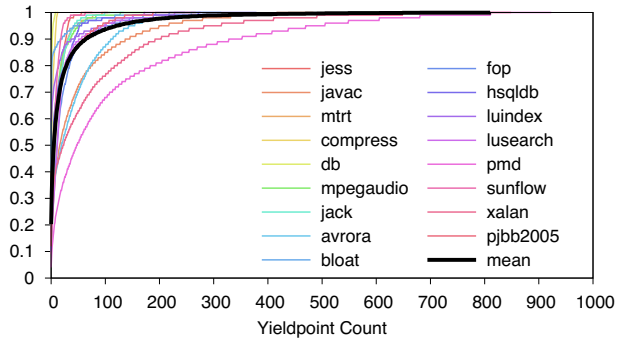


Figure 2. Cumulative frequency distribution of dynamic yieldpoint execution rates for each of our benchmarks.

We further identify every single yieldpoint inserted by the compiler and maintain an execution count for each, and Figure 2 shows the cumulative frequency of yieldpoint execution across different benchmarks. The figure suggests that among more than 35 k yieldpoints inserted by the compiler for each benchmark, just a few hundred account for most executions. On average, just 315 yieldpoints (∼1 %) account for 99 % of all the yieldpoint executions, dynamically, and 681 yieldpoints (∼2 %) account for 99.9 % of all executions. In the worst case (xalan), the same percentile is 849, which is still a tiny fraction of the static yieldpoint count. This result is interesting because it suggests that, despite the pervasiveness of yieldpoint insertion and execution, less than 1 % of yieldpoints dominate the behavior. This can possibly be exploited for a more optimized yieldpoint design as will be discussed in Section 3.3.

Summarizing, Figure 1 shows that: (*a*) yieldpoints are executed at a very high frequency, (*b*) they are relatively rarely taken, and (*c*) that sampling for FDO dominates garbage collection and lock revocation as a reason for yieldpoints to be taken, while Figure 2 shows that a tiny fraction of yieldpoints dominate execution.

2.3 Related Work

To the best of our knowledge, despite their importance to language behavior and performance, no prior work has conducted a detailed study of yieldpoint design and implementation.

Agesen [1] focuses purely on mechansism for GC-safe points, comparing an unconditional store to a guard page ('*polling*') with a *code patching* mechanism on SPARC machines. His code patching mechanism injects noop instructions to replace *all* yieldpoints. To trigger the yield, the runtime system patches every yieldpoint site, replacing the noop instructions at each site with a call. Agesen used a set of benchmarks comprising SPECjvm98, SPECjvm98 candidates, and two non-trival multi-threaded benchmarks. He reported that code patching for SPARC has a 6.6 % higher space cost than an unconditional store, on average, but delivers a 4.8 % speedup. We evaluate this design point on modern hardware and show that code patching costs dominate.

Our work differs from this prior work in multiple ways. First, we provide a detailed categorization of generalized *yieldpoint* mechanisms suited to a variety of purposes in modern run-time systems. We consider garbage collection as one use of yieldpoints, among others. The two implementations of GC-safe points measured by Agesen [1] are what we call a *global store trap-based yieldpoint* and *global code patching yieldpoint*. Second, our methodology allows us to evaluate different yieldpoint implementations over a baseline that has no injected yieldpoints. This allows us to understand the performance overheads for each configuration. In contrast, the previous work evaluated two implementations against each other with no baseline. Our selection of benchmarks is more mature, and contains a set of real-world multi-threaded applications. Since yieldpoints are naturally designed for multi-threaded contexts, our benchmark choice enables studies such as per-thread yield latency and worst-case yield latency, which are important for real-time and concurrent garbage collection. Third, we identify code patching as an optimization over other yieldpoint designs. Finally, the previous work was evaluated on venerable SPARC machines of more than fifteen years ago: what was true then may not be true now. Our experiments evaluate and report for contemporary hardware.

Click et al. [9] distinguish *GC-safe points* and *checkpoints* in their work related to pauseless GC algorithms. GC-safe points are the managed code locations where there is precise knowledge about the contents of registers and stacks, while checkpoints are synchronization locations for all mutator threads to perform some action. Our paper projects a more detailed categorization of yieldpoints and their implementations.

Stichnoth et al. [17] proposed an interesting alternative to the compiler injected yieldpoints discussed here. They focus on maintaining comprehensive GC maps that cover all managed code instruction locations so as to allow garbage collection to occur at any location without the need for designated yieldpoints. They report significant overhead for the resulting GC maps (up to 20% of generated code size) even after efforts to compress the maps. This may not be desirable in practice, so compiler-injected yieldpoints are widely used in language implementations.

3. Yieldpoints

In this section we categorize different implementations of *compiler injected yieldpoints* and describe the use of code patching as an optimization. Our focus is the use of yieldpoints in managed language implementations, where applications must yield occasionally to service run-time system requests. A given yieldpoint may be associated with compiler-generated information that records GC stack maps, variable liveness, etc. As an alternative to compiler injected yieldpoints, non-cooperative systems that do not rely on compiler support may use operating system signals to interrupt a native thread to 'yield' at arbitrary program locations [8]. This approach injects no code in the application, and only requires a signal handler to deal with the interrupt. However, the run-time system can make no assumptions about where the yield occurs, and this further prevents any useful information to be associated with the yielding location (such as stack maps for exact GC). For managed runtime systems it is much more desirable to be able to exploit such information, so we exclude the non-cooperative techniques from our categorization and focus on discussing compiler injected yieldpoints for managed language run-time systems.

3.1 Mechanisms

Because yieldpoints are frequently executed and seldom triggered, the common implementation pattern is to use the *fast-path/slow-path idiom*. The fast-path is pervasively inserted into managed application code, and does a quick check to decide whether there is any incoming request. If there is, the yieldpoint is taken and control flow goes to the slow-path which further decodes the request, and reacts accordingly. If the yieldpoint is not taken then execution continues at the next application instruction. Control transfer to the slow-path may be via a direct or indirect conditional branch, or by having the fast-path trigger a hardware trap that can be fielded by a matching trap handler.

Conditional Polling Yieldpoints This yieldpoint implementation involves a condition variable. The compiler injects a constant comparison against the value of the variable and a conditional jump to the slow path on true. In normal

```
1 .yieldpoint:
2         cmp 0 [TLS_REG + offset]
3         jne call_yieldpoint
4 .normal_code:
5         ...
```

(a) Conditional

```
1 .yieldpoint:
2         test 0 [TLS_REG + offset]
3 .normal_code:
4         ...
```

(b) Trap-based Load

```
1 .yieldpoint:
2         mov 0 [TLS_REG + offset]
3 .normal_code:
4         ...
```

(c) Trap-based Store

Figure 3. Thread-local polling yieldpoints

cases, the condition is not met, and the jump falls through to the next instruction. When the yieldpoint is enabled the jump transfers control to the slow path to execute the yield. Figure 3a shows the fast-path implementation for conditional polling yieldpoints. Jikes RVM uses this mechanism [2].

One advantage of conditional polling yieldpoints is that they provide flexibility and allow easy implementations of yieldpoints for a finer *scope*. The compiler can generate different conditional comparison instructions for yieldpoints at various locations, and at run time the variable can be set to different values to allow a subset of the conditional comparisons to be triggered, so that only a subset of yieldpoints can be taken. For example, the compiler emits `cmp [offset] 0; jne call_yieldpoint;` for Group A and `cmp [offset] 0; jgt call_yieldpoint;` for Group B. At run-time, if the conditional variable is set to -1, then only Group B takes the yieldpoints.

Moreover, the condition variable can be held in a thread-local variable, allowing yieldpoints to trigger only for particular threads.

Trap-Based Polling Yieldpoints This yieldpoint implementation involves a dedicated memory page that can be protected as appropriate. The compiler injects an access (read or write) to the page as the yieldpoint fast-path (see Figures 3b and 3c). In the common case the access succeeds and will not trigger the yieldpoint. Enabling the yieldpoints is simply a matter of protecting the page (from read or write as appropriate) to make the yieldpoint instruction generate a trap. Here the slow path is the handler used to field the trap. A load yieldpoint on x86 can be implemented as a `cmp`, or `test` to avoid the use of a scratch register. The store implementation can be exploited to store useful profiling information such as the address of the currently executing method, or the address of the yieldpoint instruction itself. The Hotspot VM uses trap-based load yieldpoints on a global protected page [13].

Once again, the access can be to a page held in a thread-local variable, allowing yieldpoints to trigger only for particular threads.

Code Patching Yieldpoints Besides the polling mechanisms described above, code patching is another possible mechanism to implement yieldpoints. A common use is *`NOP` patching*. The compiler injects several bytes of `NOP`s at yieldpoint locations, which makes no meaningful change in the generated application code. To trigger a yieldpoint the run-time system simply iterates through the code space or a stored list of *all* yieldpoint code addresses, and patches code by replacing the `NOP`s with other instructions that cause control to flow to the yieldpoint slow path. Intuitively, this approach imposes the lowest fast-path overhead (both in terms of space and time), but enabling yieldpoints is costly. Agesen [1] reported the use of this approach in 1998, and found it faster than conditional polling on a SPARC machine of that era. Our evaluation on modern hardware shows that the cost of patching the instructions dominates any potential advantage. A similar mechanism is often used for *watchpoints*, which we consider a finer-grained subtype of yieldpoints — watchpoints can be turned on and off per group, as will be discussed below.

3.2 Scope

Besides categorizing yieldpoints from the perspective of implementing mechanisms, we also categorize yieldpoints by different levels of scope. From coarser to finer levels, we discuss three scopes: *global*, *thread-local*, and *group-based*.

Global Yieldpoints These are turned on and off all at once to trigger a global synchronization of all application threads. Global yieldpoints are useful for global events such as stop-the-world GC. Yieldpoints of this scope can be implemented with different mechanisms: using a global conditional variable, a single global protected page, or an indiscriminate pass of patching through the whole code space.

Thread-Local Yieldpoints These can be turned on and off for a single thread or a group of threads. They can be used for global synchronization if the targeted threads include all the running threads. Yieldpoints of this scope are useful for targeted per-thread events, such as pair handshakes between two threads. Yet it provides flexibility as they can also be used for global events. As noted above, using a thread-local condition variable or thread-local protected page enables thread-local conditional polling or trap-based polling, respectively. However, there is no straight-forward implementation of a thread-local unconditional code patching yieldpoint [1], since there is no easy guarantee of the patched code being executed only by certain threads.

Group-based Yieldpoints These are grouped, and can be turned on and off by group. They are also known as 'watchpoints' [3]. This type is useful as guards for speculative execution. For example, in places where the compiler makes an assumption regarding type specialization or an inlining decision, it inserts a group-based yieldpoint before the specialized or inlined code. Whenever the run-time system notices that the assumption breaks, it enables that group of yieldpoints to prohibit further execution of the code under false assumption. Code that reaches the enabled yieldpoints will take a slow-path, where the run-time compiler can make amends and generate new valid code. Code patching is the most straight-forward mechanism to implement group-based yieldpoints, since it naturally needs to know the offset of each yieldpoint. To adapt to group-based scope, it simply records and patches yieldpoint addresses by group. Conditional polling also fits well in group-based scope by using different conditional variables or different conditions per group. Trap-based polling only works for a limited number of groups as there are limited trap signals and protected pages.

3.3 Code Patching As An Optimization

We note that the unconditional code patching yieldpoint presents a severe tradeoff. The common case cost of a noop-patched yieldpoint is very close to zero. However, we measured the cost of patching and found that when patching is performed at every timer tick, it adds on average 13.4 % overhead when all yieldpoints are patched. We then measured the effect as we reduced the number of yieldpoints patched, and found that it fell to 0.6 % when 681 ($\sim$99.9 % in Figure 2) are patched and just 0.3 % when 315 ($\sim$99 %) are patched. This observation led us to consider code patching as a possible optimization over conditional or trap-based yieldpoints.

When used as an optimization, code patching selectively overwrites only the most frequently executed yieldpoints with no-ops. When a yieldpoint is triggered, the optimized yieldpoints are rewritten to their original state (or to unconditional yields). Once the yield is complete, the most frequently executed yieldpoints are once again elided. The choice of which yieldpoints to optimize will depend on the cost-benefit tradeoff between the patching cost and the cost of executing the unoptimized yieldpoint. If the 300 or so most heavily executed yieldpoints could be sucessfully identified and patched, it seems possible that the optimization would be almost entirely effective and yet introduce only a tiny overhead due to patching. Possible refinements to this optimization include parallelizing the code patching (also applicable to the code patching yieldpoint), aborting patching if the yield succeeds before all are patched, and ordering the patching so that the most frequently executed yieldpoints are patched first. We conduct a preliminary evaluation of code patching as an optimization. Although at the time of publication we had not succesfully demonstrated code patching as an optimization, we believe that the analysis we present here is encouraging.

Summary In this paper, we evaluate global and thread-local versions of polling yieldpoints: *i.e.*, [Global, Thread-Local]×[Conditional, Trap-based Load, Trap-based Store] as they are most relevant to global run-time synchronization events such as garbage collection. We also include the cost of the the fast-path of code patching yieldpoints in our evaluation, which is several bytes of noop.

4. Methodology

In this section, we present the software, hardware and measurement methodologies we use. We base our methodology on similar work introduced by Yang et al. [18], adapting it to the task of measuring yieldpoints. The principal methodological contribution of this paper is an **omitted yieldpoint** methodology, which allows us to use a system with *no injected yieldpoints* as a baseline. We describe the omitted yieldpoint methodology below.

Measurement Methodology We implement all yieldpoints in version 3.13 of Jikes RVM [2], with a production configuration that uses a stop-the-world generational Immix [5] collector. We hold heap size constant for each benchmark, but because our focus is not the performance of the garbage collector itself, we use a generous 6× minimal heap size for each benchmark with a fixed 32 MB nursery.

We use Jikes RVM's *warmup replay* methodology to remove the non-determinism from the adaptive optimization system. Note that the use of replay compilation has the important benefit of obviating the need for the adaptive optimization system to perform profiling, which would otherwise make our *omitted yieldpoints* methodology impossible. Before running any experiment, we first gather compiler optimization profiles from the best performance run from a set of runs for each benchmark. Then, when we run the experiments, every benchmark first goes through a complete run to warm up the run-time (allowing all the classloading and method resolving work to be done), and then the compiler uses the pre-collected optimization profiles to compile benchmarks and disallows further recompilation. This methodology greatly reduces non-determinism from the adaptive optimizing compiler. Note that we use the replay advice from the status quo build. However, since our different builds impose little change in the run-time system, we expect the bias introduced by using the same advice to be minimal as well.

Omitted Yieldpoint Methodology To evaluate the overhead of various yieldpoint implementations, we developed a methodology with *no* injected yieldpoints, which served as a baseline against which each of the yieldpoint implementations could be compared. The methodology depends on two insights. First, we can disable two of the three systems that depend on yieldpoints: sampling for feedback-directed

optimization, and lock revocation for biased locking. As mentioned above, the warmup replay methodology provides a sound basis for empirical analysis such as this, and happens to have the side effect of not requiring sampling for FDO. Biased locking is an optimization that we can readily disable, removing the need for lock revocation at the cost of modest performance losses on some multi-threaded benchmarks. Second, *explicit* yieldpoints remain in place, even when we disable *injected* yieldpoints. Empirically, explicit yieldpoints are sufficiently frequent that garbage collection — the one remaining component dependent on yieldpoints — can occur in a timely manner. We quantify the slop that removal of injected yieldpoints adds to reaching explicit GC-safe points by measuring the time taken for threads to yield and comparing it with total mutator time. The average effect on mutator time due to slower latency to reach explicit GC-safe points is 0.9 %, which is mostly due to one benchmark which triggers GC very frequently (lusearch, 9.1 %). For a fair comparison, in our measurements, injected yieldpoints have an empty slow path and will not bring the thread to a GC-safe point so that GC always relies on explicit yieldpoints, and the slightly longer GC-safe point latency persists for all the experiments. The obvious alternative to our approach would be to remove the need for garbage collection altogether by using a sufficiently large heap. However, this would be impractical for benchmarks such as lusearch which allocate prolifically, and would measurably degrade benchmark locality [10].

Hardware and Software Environment Our principal experiments are conducted on a recent 22 nm Intel Core i7 4770 processor (Haswell, 3.4 GHz) with 8 GB of 1600 MHz DDR3 RAM. To evaluate the impact of microarchitecture, we also use a 32 nm Intel 2600 Core i7 2600 processor (Sandy Bridge, 3.4 GHz) with 8 GB of 1333 MHz DDR3 RAM. Aside from the difference in Haswell and Sandy Bridge microarchitectures and memory speeds, the machines are extremely similar in their specifications and configuration. We use Ubuntu 14.04.1 LTS server distribution running a 64 bit (x86_64) 3.13.0-32 Linux kernel on both machines.

Benchmarks We draw the benchmarks from the DaCapo suite [7], the SPECjvm98 suite [15], and pjbb2005 [6] (a fixed workload version of SPECjbb2005 [16] with 8 warehouses that executes 10 000 transactions per warehouse). We use benchmarks from both 2006-10-MR2 and 9.12 Bach releases of DaCapo to enlarge our suite and because a few 9.12 benchmarks do not execute on Jikes RVM. We exclude eclipse from the suite since our thread-local trap-based implementation requires larger space for thread-local storage, which makes eclipse run out of metadata space under the default configuration.

5. Results

We report the performance of each of the yieldpoint designs. We start by evaluating the overhead of the common *untaken* case of each of the yieldpoints. Next we evaluate the yieldpoints when they are taken with normal frequency. Finally, we measure the time-to-yield (latency) for the different yieldpoints.

5.1 Overhead of Untaken Yieldpoints

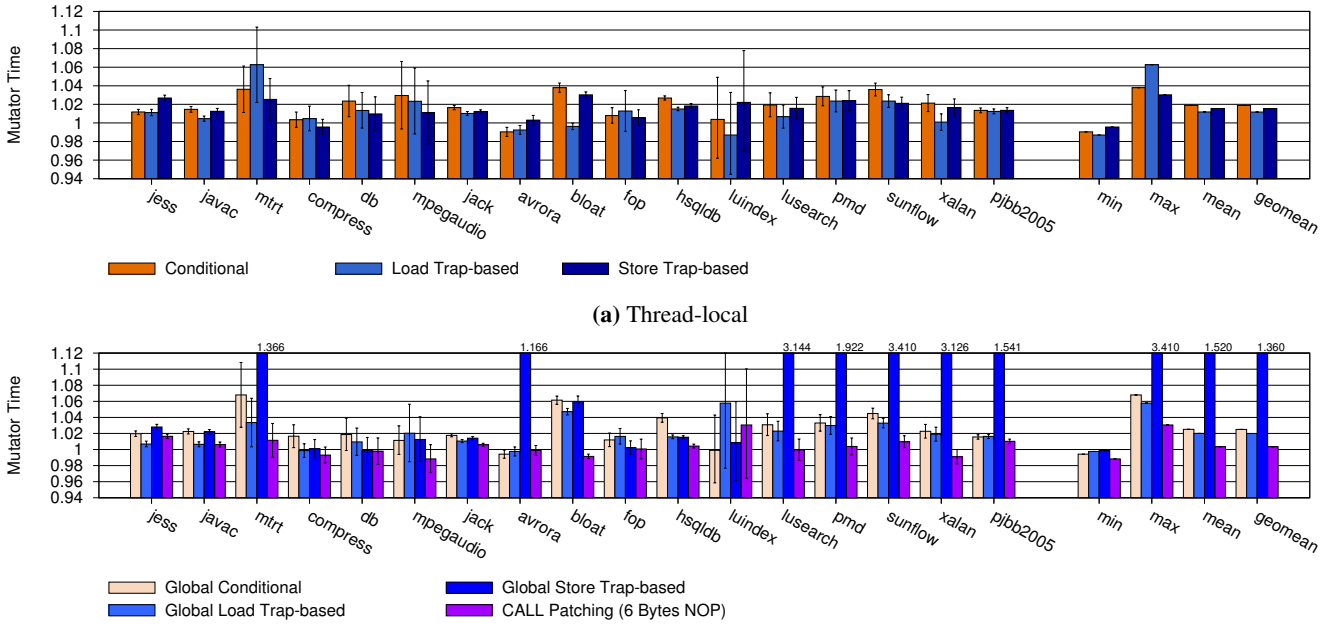
We use the *omitted yieldpoint methodology* of Section 4 to measure the impact of each yieldpoint design on mutator performance in the case where the yieldpoint is *never* taken. This reflects the common case, since as Section 2.2 showed, only about $1/20\,000$ yieldpoints are actually taken. We first evaluate the overheads for thread-local yieldpoints before considering global yieldpoint designs.

Thread-Local Yieldpoints Figure 3 shows the code for three thread-local yieldpoint designs. Figure 4a shows the overheads on the Haswell microarchitecture. The geometric mean overheads are 1.9 % for the conditional, 1.2 % for the load trap, 1.5 % for the store trap.

Our evaluation on the Sandy Bridge hardware reveals some interesting differences between microarchitectures. The geometric means for Sandy Bridge are 2.3 % for the conditional, 2.1 % for the load trap, 1.6 % for the store trap. Thus the conditional and trap-based yieldpoints are noticeably higher and more homogenous on the older machine. Interestingly, the load trap yieldpoint is significantly lower on the new machine, from 2.1 % down to 1.2 %, while the improvements brought by the newer microarchitecture on other implementations are marginal. This result highlights the sensitivity of these mechanisms to the underlying microarchitecture, and the consequent need to reevalaute and rethink such designs in contemporary settings.

Global Yieldpoints Global yieldpoints are very similar to the thread-local yieldpoints shown in Figure 3, only rather than referring to thread-local storage (via the `TLS_REG` in Figure 3), they refer to a single global value for the conditional yieldpoint and a single global guard page for the trap yieldpoint. Figure 4b shows the overheads for the global yieldpoints on the Haswell microarchitecture. The geometric mean overheads are 2.5 % for the conditional, 2.0 % for the load trap, 36 % for the store trap! Each of these are higher than their thread-local counterpart. The difference between the local and global yieldpoints is moderate for the conditional and the load trap (respectively 0.6 % and 0.8 %). But for the store trap, the slowdown is extreme. The reason is obvious. It is clear from Figure 4b that all multi-threaded benchmarks account for much of the increase in store trap overhead. This is due to write contention on the guard page caused by multiple user threads trying to write to the same cache line. These results make it clear that aside from the additional flexibility offered by thread-local yieldpoints, they also offer substantially lower overheads.

We also measured the six-byte noop overhead, which acts as the fast-path of one implementation of code patching yieldpoint. The noops can be patched into an absolute



(a) Thread-local

(b) Global

Figure 4. Mutator overhead of *untaken* thread-local and global yieldpoints on the Haswell microarchitecture. The graph shows times normalized to the *no yieldpoint* baseline. The geometric mean overheads for the thread-local yieldpoints are 1.9 % for the conditional, 1.2 % for the load trap, 1.5 % for the store trap. The global yieldpoints suffer systematically higher overheads due to cache contention while the six-byte noop as code patching fast-path imposes minimal overheads (0.3 %).

call instruction on demand. The six-byte noop has the least overhead among all the yieldpoints we measured, (0.3 %) on Haswell, and zero measurable overhead on Sandy Bridge. This only reflects the common case fast-path, it does not include the cost of performing code patching.

These results indicate a number of interesting findings. First, the conditional yieldpoint does have a reasonably low overhead but nonetheless is the worst performing among untaken thread-local yieldpoints. Second, the overhead of the code patching yieldpoint in the untaken case is (perhaps unsurprisingly) very low.

5.2 The Overhead of Taken Yieldpoints

In the previous section we looked at the overheads due to yieldpoints when they are *never* taken. In practice, of course, yieldpoints are taken, even if rarely. We now extend the same methodology as above, only that we allow yieldpoints to be triggered normally by the underlying profiling system that dominates yieldpoint activity (Figure 1). However, we implement an *empty* slow path activity: when the yieldpoint takes its slow path, we simply turn off the yieldpoint and return to the mutator rather than actually undertake profiling or any other task. Notice that in the case of the conditional yieldpoint, this means that there is very little additional overhead, whereas in the trap-based yieldpoints, the trap must still be taken and serviced before returning.

Figure 5 shows the results for the Haswell microarchitecture. The geometric mean overheads for the yieldpoints are 1.9 % for the conditional, 2.7 % for the load trap, 3.0 % for the store trap. Notice that the total overheads are now dramatically evened out compared to the *untaken* results seen in Figure 4. The conditional has non-measurable extra overhead for taking yieldpoints. However, though clearly faster than conditional yieldpoints in untaken cases, trap-based yieldpoints are now slower due to the overhead associated with servicing the traps. These results undermine the advantage of load and store trap-based yieldpoints when yieldpoints are required to be taken frequently.

5.3 Time-To-Yield Latency for GC

A third performance dimension for yieldpoint implementations is the time it takes for *all* mutator threads to reach a yieldpoint. This is of course dominated by the *last* thread to come to a yieldpoint. Intuitively, a trap-based yieldpoint will perform worse on this metric than a conditionally polling yieldpoint because it is subject to the vagaries of the operating system's servicing the trap and any scheduling perturbations that may induce. We measure the time-to-yield latency

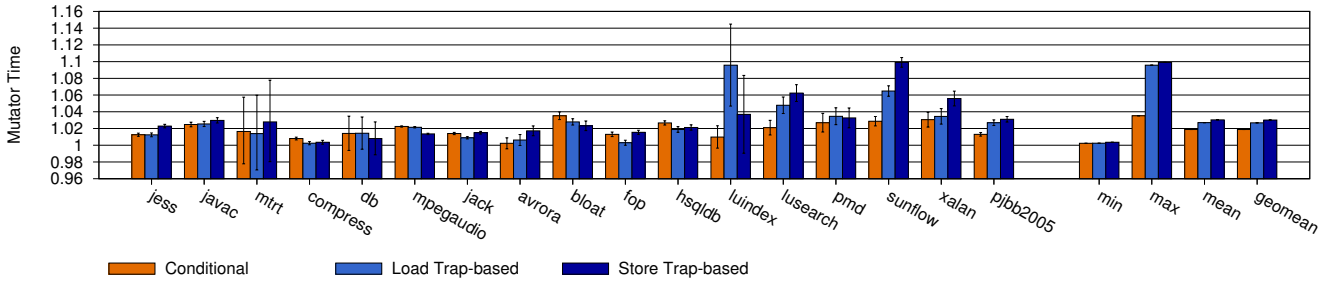


Figure 5. Mutator overhead of *sometimes taken* yieldpoints (thread-local) on the Haswell microarchitecture. The graph shows times normalized to the *no yieldpoints* baseline. The geometric mean overheads for the yieldpoints are 1.9 % for the conditional, 2.7 % for the load trap, 3.0 % for the store trap.

for each GC by using thread-local polling yieldpoints with multi-threaded benchmarks.[1]

Figure 6 shows the thread yield latency (in cycles) for each GC. Every point in the figure shows the latency from when the collector initiates the yield to when *each* thread reaches a yieldpoint. The horizontal line indicates the 95th percentile among the data points (278 k cycles, 665 k cycles and 659 k cycles, respectively, for the three implementations on Haswell). Conditional polling has a substantially lower average time-to-yield, but is also more tightly grouped. This is unsurprising, since trap-based implementations require a system call to protect the polling page, and require signal handling to take yieldpoints, while conditional polling involves a simple change on the value of the polling flag, and a call.

Figure 7 shows the distribution of *worst-case* thread yield latency across all of our multi-threaded benchmarks. Worst-case thread yield latency is the time from when the collector initiates the yield to when the *last* thread reaches a yieldpoint. We can see that on both machines, conditional polling has a much tighter distribution and lower latency. We examined the worst (rightmost) results in each scenario and found that the majority are from two benchmarks sunflow and lusearch. 15 out of the worst 30 results are from sunflow and 10 out of 30 are from lusearch. For the best-case time-to-yield latency (*i.e.*, the fastest time from GC initiation to all threads yielding) there is a clear distinction between conditional and trap-based polling. On Haswell, conditional polling has the lowest yield latency of 109 k cycles while trap-based polling is 174 k cycles for both load and store.

From these measurements, we see that conditional polling yieldpoints have a markedly better time-to-yield latency than trap-based yieldpoints on average and at the 95th percentile. However, the worst-case time-to-yield latency is not well correlated with yieldpoint implementation, but rather affected by the operating system and the benchmarks.

[1] For single-threaded benchmarks the only application thread yields immediately on a failed allocation, so latency is not affecetd by the particular yieldpoint implementation, so we exclude the single-threaded benchmarks.

6. Conclusion

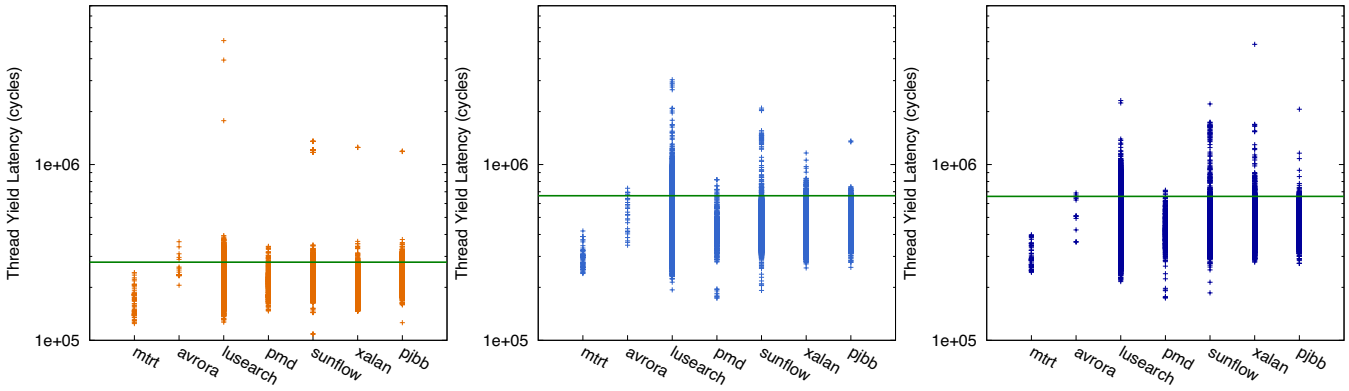
Yieldpoints are a principal mechanism for determining when a managed language thread must yield. They are used for GC-safe points at which the collector can safely sample mutator state, for revocation of biased locks, on-stack replacement, and for lightweight profiling used by feedback directed optimizations. Because application threads must be responsive to such requests, it is important that the period between yieldpoint executions be well bounded and that they execute with low overhead in the common fast-path case when there is no request to service.

We have identified and evaluated a range of yieldpoint mechanisms. We find that the tradeoff between common-case fast path execution and overheads in the uncommon case can be severe. While an unconditional trap-based poll has low overhead in the common case, it is costly when the yield occurs, resulting in slightly worse performance than a simple conditional test, on average. An unconditional code patching yieldpoint presents an even move extreme tradeoff, with near zero common case overhead but substantial patching overheads at every yield. We highlight the microarchitectural sensitivity of these mechanisms, indicating the need for virtual machine implementors to reassess their performance assumptions periodically. We also identify that code patching presents an interesting opportunity for an optimization, replacing a few of the most frequently executed yieldpoints with noops at times when yields are not required.

Overall we found that conditional polling has the most desirable characteristics: low overhead, fast time-to-yield, and implementation simplicity.

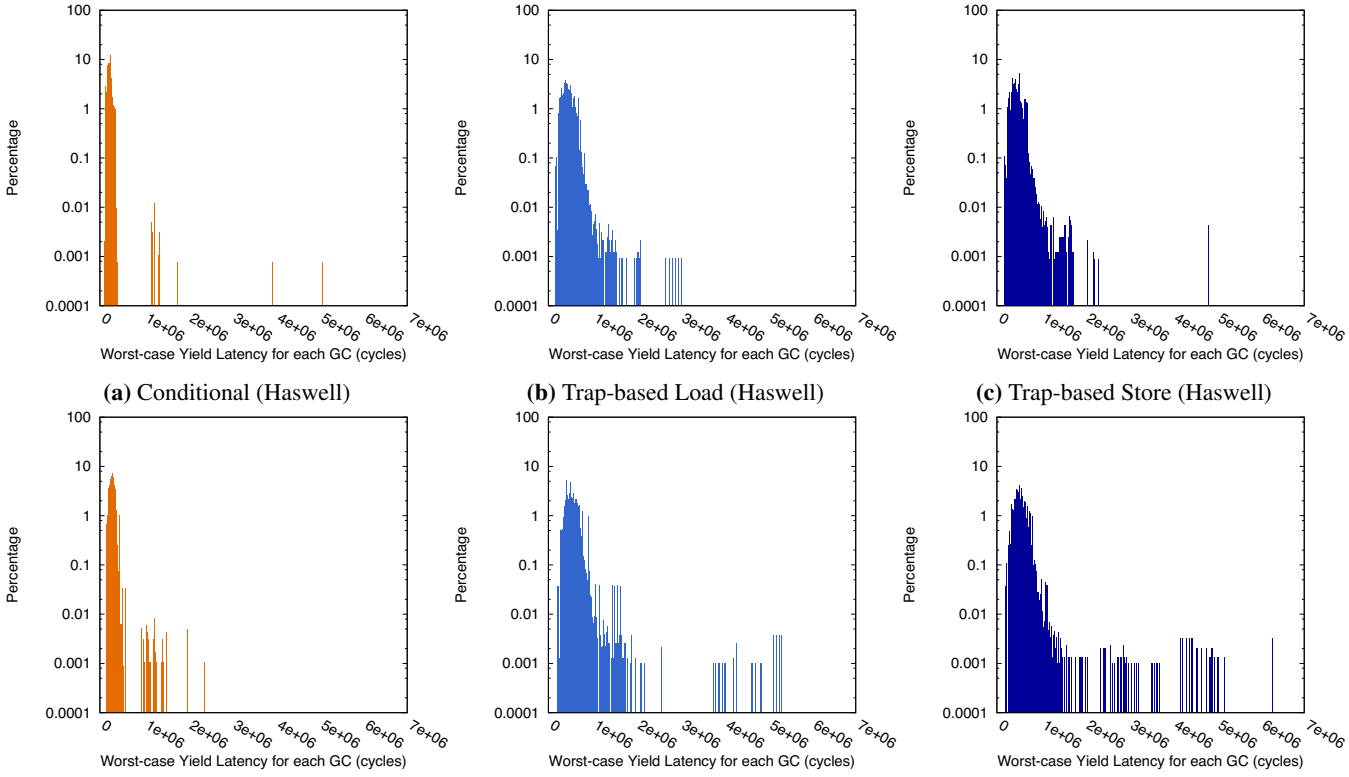
Acknowledgments

This work is supported by the Australian Research Council under grant ARC DP140103878, and the National Science Foundation under grants nos. CNS-1161237 and CCF-1408896. NICTA is funded by the Australian Government



(a) Conditional **(b)** Trap-based Load **(c)** Trap-based Store

Figure 6. Time-to-yield latency for polling yieldpoints, measured in cycles (log-scale y-axis), for each of the multi-threaded benchmarks.



(a) Conditional (Haswell) **(b)** Trap-based Load (Haswell) **(c)** Trap-based Store (Haswell)

(d) Conditional (Sandy Bridge) **(e)** Trap-based Load (Sandy Bridge) **(f)** Trap-based Store (Sandy Bridge)

Figure 7. Time-to-yield worst-case latency distribution for each GC. The conditional yieldpoint has a much tighter distribution, and the newer Haswell microarchitecture produces tighter distributions than its older Sandy Bridge counterpart.

through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

[1] O. Agesen. GC points in a threaded environment. Technical report, Sun Microsystems Laboratories, Palo Alto, California, 1998. URL http://dl.acm.org/citation.cfm?id=974974.

[2] 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 *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Denver, Colorado, 1999. doi: 10.1145/320384.320418.

[3] Apple. WebKit JavaScript Core. URL http://trac.webkit.org/wiki/JavaScriptCore.

[4] M. Arnold and D. Grove. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2005. doi: 10.1109/CGO.2005.9.

[5] S. M. Blackburn and K. S. McKinley. Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Tucson, Arizona, 2008. doi: 10.1145/1375581.1375586.

[6] S. M. Blackburn, M. Hirzel, R. Garner, and D. Stefanović. pjbb2005: The pseudoJBB benchmark. URL http://users.cecs.anu.edu.au/~steveb/research/research-infrastructure/pjbb2005.

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, 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, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Portland, Oregon, 2006. doi: 10.1145/1167515.1167488.

[8] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9), Sept. 1988. doi: 10.1002/spe.4380180902.

[9] C. Click, G. Tene, and M. Wolf. The pauseless GC algorithm. In *ACM/USENIX International Conference on Virtual Execution Environments*, Chicago, Illinoois, 2005. doi: 10.1145/1064979.1064988.

[10] X. Huang, S. M. Blackburn, K. S. McKinley, J. Moss, Z. Wang, and P. Cheng. The garbage collection advantage: Improving program locality. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, Vancouver, Canada, 2004. doi: 10.1145/1028976.1028983.

[11] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook*. Chapman & Hall, 2012.

[12] V. Kumar, D. Frampton, S. M. Blackburn, D. Grove, and O. Tardieu. Work-stealing without the baggage. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, Tucson, Arizona, Oct. 2012. doi: 10.1145/2384616.2384639.

[13] OpenJDK Group. Hotspot VM. URL http://openjdk.java.net/groups/hotspot/.

[14] F. Pizlo, D. Frampton, and A. L. Hosking. Fine-grained adaptive biased locking. In *International Conference on Principles and Practice of Programming in Java*, Kongens Lyngby, Denmark, 2011. doi: 10.1145/2093157.2093184.

[15] SPEC. SPECjvm98, release 1.03, 1999. URL http://www.spec.org/jvm98.

[16] SPECjbb2000. SPECjbb2000 (Java Business Benchmark) documentation. URL https://www.spec.org/jbb2005/.

[17] J. M. Stichnoth, G.-Y. Lueh, and M. Cierniak. Support for garbage collection at every instruction in a Java compiler. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Atlanta, Georgia, 1999. doi: 10.1145/301618.301652.

[18] X. Yang, S. M. Blackburn, D. Frampton, and A. L. Hosking. Barriers reconsidered, friendlier still! In *ACM SIGPLAN International Symposium on Memory Management*, Beijing, China, June 2012. doi: 10.1145/2258996.2259004.

[19] X. Yang, S. M. Blackburn, and K. S. McKinley. Computer performance microscopy with SHIM. In *International Symposium on Computer Architecture*, Portland, Oregon, 2015. doi: 10.1145/2749469.2750401.