Path Specialization: Reducing Phased Execution Overheads

Filip Pizlo *
Purdue University
West Lafayette, IN 47907
USA
pizlo@purdue.edu

Erez Petrank †
Microsoft Research
One Microsoft Way
Redmond, WA 98052
USA
erez@cs.technion.ac.il

Bjarne Steensgaard
Microsoft Research
One Microsoft Way
Redmond, WA 98052
USA
Bjarne.Steensgaard@microsoft.com

Abstract

As garbage collected languages become widely used, the quest for reducing collection overheads becomes essential. In this paper, we propose a compiler optimization called *path specialization* that shrinks the cost of memory barriers for a wide variety of garbage collectors including concurrent, incremental, and real-time collectors. Path specialization provides a non-trivial decrease in write-barrier overheads and a drastic reduction of read-barrier overheads. It is effective when used with collectors that go through various phases each employing a different barrier behavior, and is most effective for collectors that have an idle phase, in which no barrier activity is required. We have implemented path specialization in the Bartok compiler and runtime for C# and tested it with state-of-the-art concurrent and real-time collectors, demonstrating its efficacy.

Categories and Subject Descriptors D.1.5 [*Object-oriented Programming*]: Memory Management; D.3.3 [*Language Constructs and Features*]: Dynamic storage management; D.3.4 [*Processors*]: Memory management (garbage collection); D.4.2 [*Storage Management*]: Garbage Collection

General Terms Algorithms, Design, Performance, Reliability

1. Introduction

Garbage collection is widely acknowledged for speeding up software development while increasing security and reliability. Garbage-collection has been incorporated into modern popular languages such as C# and Java. Recent advances in the development of advanced garbage collectors, such as concurrent (non-intrusive), parallel, and real-time collectors have further increased garbage collection use in modern computing. The popularity that garbage collected environments are gaining makes the task of reducing garbage collection overheads highly desirable.

In this work, we propose a method for optimizing a variety of garbage collectors, without any need for system or hardware support. Our method is aimed at reducing the overheads induced by memory barriers and is operated by modifying the compiler's code-generation component. Memory barriers induce high overheads ranging up to 60%, depending on the platform used, the function the barriers perform for the memory management component, and the benchmark [6, 23]. Memory barriers of many collectors change their behaviors according to phases in the execution of the collector. In particular, many collectors employ a very efficient barrier (or no barrier at all) while they are idle, and a more demanding barrier operation while active. Selecting the mode of operation cannot be done at compile time, because the times at which the collector is idle and active are determined only during program execution. Therefore, such barriers contain code that determines which operations to execute according to the collector phases. In particular, checks are executed even when the barrier is not required to do anything useful in the current collector phase.

We propose to use path specialization in order to reduce the barrier overheads by reducing the amount of work required for phase checks. This reduces the amount of computation required, allows better register allocation, reduces the load on branch prediction resources, and improves code cache behavior. Path specialization starts with code cloning, creating multiple copies of the program code, and then modifying each copy to handle one or more phases of the collector. Since each specialized code version is restricted to being executed in only a subset of the possible phases, the checks it has to perform in order to determine the current phase and what to do in this phase can be reduced. In particular, if a specialized code version is only executed in the idle phase, for which no barrier action is required, the barrier code can be entirely eliminated in this specialized code version.

It remains to specify how the control is transferred from one copy of the code to another. Timely transfer of control is crucial, because threads must respond to phase changes of the garbage collector in a timely manner, allowing the collector to perform its task. This causes transfers between specialized paths to become more complicated. We propose a simple method for code generation of appropriate control transfer that avoids the use of specialized static analysis. The modifications to the compiler and runtime that are needed by the the proposed path specialization method are minimal.

Using code duplication, specialization and transferring control back and forth between various specialized versions of the input code is a technique that was also used in a different context for bursty program tracing or sampling [1, 16]. Our method can be viewed as importing and adapting these techniques to the memory management world to achieve an important overall reduction on memory barrier overheads. In addition, we propose a simple compiler technique for reducing the code size increase without requiring a specialized static analysis.

* Work done while the author was an intern at Microsoft Research.

† Work done while the author was on sabbatical leave from the Computer Science Department, Technion, Haifa, Israel.

ISMM'08, June 7–8, 2008, Tucson, Arizona, USA.

We have implemented path specialization in the Bartok compiler and runtime, and tested it with several important garbage collectors. Our measurements show a shrinkage of the barrier costs with dramatic improvements of up to 45% in overall application execution time for collectors that employ a read-barrier and nice improvements of up to 5% for collectors that employ a write-barrier.

Our contributions can be summarized as follows.

1. A proposed optimization of memory barriers for memory management that is suitable for a variety of garbage collectors, and maybe usable for other overheads in various systems.
2. A simple method for reducing the code size increase. This method can be easily implemented, it does not require a specialized compiler analysis, and it is very effective.
3. An implementation and measurements of the proposed method with several relevant memory barriers, employed by state-of-the-art garbage collectors.

Although we propose and examine these techniques in the context of reducing the overheads of garbage-collected environments, there may be an interesting potential for pushing these techniques and generalizing them further for other overheads added with modern language implementations and measurements, e.g., trace monitors.

Organization. We start with a short background on garbage collectors and memory barriers in Section 2. In Section 3 we describe the path specialization basics concentrating on intra-procedural optimization. Inter-procedural and further optimizations are discussed in Section 4. The implementation is described in Section 5 and measurements are reported in Section 6. Related work is discussed in Section 7 and we conclude in Section 8

2. Garbage Collection Barriers and Phases

Many garbage collectors, such as concurrent, incremental, and real-time collectors, require that special code is executed during pointer modification or load. Some collectors even incur an overhead on any memory access (not only for pointers). Such code is called a memory barrier. A *write-barrier* is a piece of code that is executed with each write (usually this refers to pointer modifications only) and a *read-barrier* is a piece of code that is executed during each memory load.

Memory barriers are required for different reasons. For example, concurrent mark-sweep garbage collectors trace the set of reachable objects, while the program threads concurrently modify pointers in them. Unless compensated for, such pointer changes may foil the trace of the heap yielding wrong conclusions about the set of reachable objects. A write-barrier is typically used by concurrent and incremental collectors to allow cooperation between the program and the collector and guarantee that the collector correctly identifies all live objects. Such cooperation between the program and the collector is only needed during the collector's tracing phase.

Each program thread must cooperate with the collector according to the collector phase. Unless the program threads are halted to simultaneously acknowledge a phase change, the various threads notice a collector phase change at different times. On-the-fly collectors allow such latitude in threads' cooperation in order to avoid halting all threads simultaneously. The points in execution in which threads must notice a phase change are called *safe-points* or *GC-points*. Path specialization attempts to take advantage of this latitude by checking the phase once in a safe-point and then avoid repeated checking on each memory barrier.

Real-time collectors typically employ a compacting mechanism in order to avoid unexpected fragmentation. To preserve real-time properties, the compaction is run incrementally [4, 3] or concurrently [17, 9, 23]. Often, such concurrent copying requires a read-barrier to be introduced. The cumulative cost of a read-barrier is typically higher than that of a write-barrier, and therefore our method often shows even more drastic improvements when applied to collectors that employ a read-barrier. Many incremental and concurrent collectors are known today, and practically all of them use phases and can make use of path specialization, see for example [25, 10, 20, 22, 7, 12, 11, 13, 14, 24, 9, 21, 2, 5, 23].

3. Path Specialization

Consider a collector that goes through phases and for which the behavior of a memory barrier depends on the phase. The simplest implementation, and one that is used by almost all collectors, is to start the memory barrier by checking what is the current phase and then executing the relevant barrier code for that phase. These checks require repeated computational effort, they use branch prediction resources, and they pollute the code cache. Path specialization attempts to substantially reduce the need for phase checks.

The basic idea is to clone code fragments at the IL level and then create specialized versions of the code which use barriers that are valid for only a partial set of the possible phases. Typically, one would partition the set of possible phases into disjoint subsets and create specialized versions of the code for each of these subsets. The simplest partition, and one that is sensible to use, is the one that specializes one code version for handling the idle phase and uses another version to handle all other phases. Typically, a code version that only executes the idle phase has no barrier code at all, making it very efficient.

Given the general idea of executing code specialized for garbage collector phases, we must deal with the question of how to guide the threads to use the appropriate code at all times. In practice, we cannot expect an automatic costless mechanism that switches execution from one version of the code to the other when the collector changes the phase of the execution. While dealing with this issue, we will also make an attempt to avoid excessive duplication of code, as much as possible.

Garbage collection phase changes are typically only required to be recognized by the mutator threads at specific program code locations denoted *safe-points*. Essentially, we will perform phase checks after safe-points and use the information obtained by the check in all memory barriers until the next safe-point occurs. Path specialization is most likely to be useful when memory barriers occur much more frequently than safe-points, as we move phase checks out of the barriers and towards safe-points locations. The question is how to take advantage of this in practice.

For simplicity of presentation, we start by describing a naïve strategy for transferring control from one path to the other. This strategy may require the use of static dominance analysis in order to obtain good performance. The naïve method is described in Section 3.1. Next, in Section 3.2, we describe our preferred method, which is much simpler and is easy to incorporate into a compiler, with no need for any specialized static analysis. If the compiler performs trivial dead-code elimination (and most compilers do), then it will generally perform as well as or better than the naïve method.

3.1 The Naïve Approach

Assume we have two versions of the code, each specialized for different subsets of the possible phases. In order to perform the control transfer between the two specialized code fragments, a phase check operation is added after each safe-point in both specialized code versions. Following the phase check, a conditional branch transfers the flow to the appropriate point in the appropriate specialized code. Execution continues in the chosen code version until the next safe-point or method call.

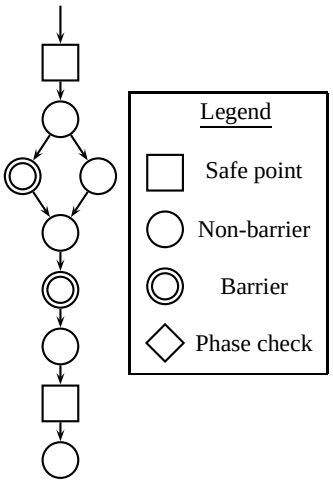


Figure 1. A simple code fragment, including instructions that run a memory barrier and instructions that don't, a conditional branch, and two safe-points.

This simple method is depicted in Figures 1 and 2(a). Figure 1 shows a program fragment with a safe-point, and then some instructions (two of them include a memory barrier) followed by a second safe-point and finally a simple subsequent instruction. The legend of the various nodes appear in this drawing. Figure 2(a) depicts the same code fragment when specialized according to the naïve method. The code is cloned and two specialized versions are created. In addition, after each safe-point there is a phase check and a jump to the appropriate specialized path.

The simple scheme above causes a significant increase in code size. Each program point in the original code will have a corresponding program point in each specialized code version. The code is expected to essentially double in size. The problem is that there is a lot of code that is duplicated without being specialized and some of this duplication can be avoided. In particular, all code that appears after a safe-point and before the next memory barrier(s) is exactly the same in both specialized code versions. The same holds for all code prior to a safe-point and after the last preceding barrier operation(s).

A simple solution for eliminating this redundant duplication is to unconditionally transfer control from all code versions into one predetermined "main" code version that executes all the code sequences which are known to be equal in all versions. In particular, before reaching any safe-point, control is transferred into the main code version, and it keeps on executing in that version until arriving at the first memory barrier. At that point, a conditional branch transfers execution to the appropriate specialized version. After the first memory barrier, execution can continue uninterrupted in all specialized versions (including the main version) until the next safe-point.

This seems simple enough. However, the execution paths between safe-points and subsequent memory barriers are not necessarily simple. It is possible that a branch appears after a safe-point and several code paths fork out before a memory barrier appears in any of them. Similarly, at a merge point of multiple paths, memory barriers may or may not have occurred on all paths from a safe point. Therefore, finding the "first" memory barriers after a given safe-point, and not adding redundant tests to memory barriers that are not "first" in the main version, requires a static analysis that resembles a dominance analysis. While this is certainly doable, a simpler approach may be preferable. Our approach will not need to compute or use previously computed information by the compiler. Instead, it will only assume that the compiler runs a dead-code elimination procedure, and treat this procedure as a black box. In what follows, we describe our preferred method that enjoys the optimization discussed above without employing any specialized static analysis.

3.2 The Path Specialization Method

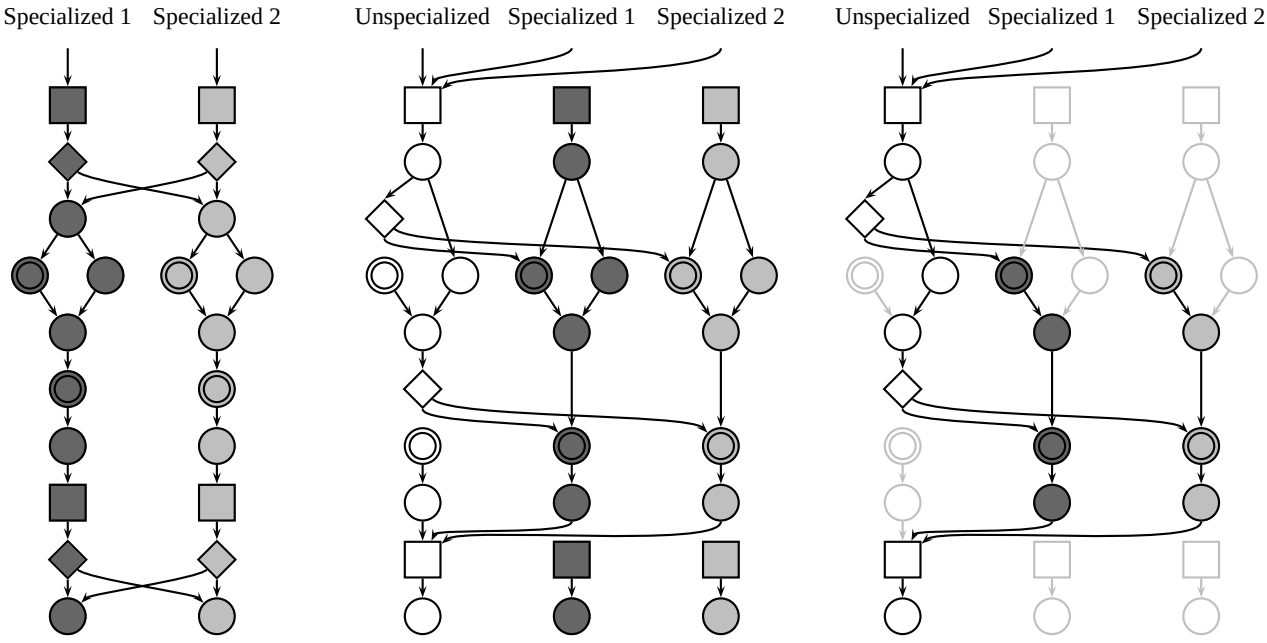
The keys ideas of the preferred path specialization method are: (1) add an additional main code version that is responsible for performing all the phase checks, (2) add a phase check before *every* barrier (not only the first one) in the main code version, (3) transfer control to the main code version near safe-points, and (4) use dead code elimination to perform the trimming of irrelevant conditional branches.

Assume as given a requirement for a memory barrier (either a read- or a write-barrier or both) which varies according to m different program phases. The program phase may dynamically change during the execution of the program and must be recognized no later than when control reaches well-defined safe-points in the program code. The first step is the partitioning of the set of m phases into any n subsets, $S_1, \ldots, S_n$, where $2 \leq n \leq m$, and design a specialized barrier for each of the subsets. On one extreme, each phase is represented by a distinct specialized version and $n = m$. On the other extreme, which is the simplest and a very effective design choice, we have $n = 2$ with one barrier version handling the idle phase (or the one with the smallest overhead) and the other barrier version handling all the other phases.

In our method of choice, we start by creating $n+1$ versions of the code. The first version (version 0) handles all common code and branches into the specialized versions of the code. This version of the code employs the original memory barriers and is called *unspecialized*. (As will become clear later, memory barriers will never be handled by this code version, so having no barriers at all will do as well.) There are n more versions of the code. The i-th version of the code contains a specialized barrier code that is specifically designed to handle only the phases that belong to the i-th subset of phases, S_i, plus additional code to transfer control back to version 0, when necessary. A decision function supplied by the developer examines the current phase and decides the number i of the code version that should be executed. The code for control transfer is automatically inserted into the code copies by the compiler, as explained below. The decision function and each of the specialized barriers to be used in the i-th specialized version of the code (for each $1 \leq i \leq n$) should be written by the garbage-collector designer.

We propose an intra-procedural path specialization, and discuss additional inter-procedural optimizations in Section 4.1. Assume we are given the code for a method and need to specialize it. The compiler starts by creating n additional extra copies of the code. In each of the n copies, the corresponding specialized barrier replaces the general barrier. In the original code, just before *any* memory barrier, we insert code that computes the decision function P and according to the resulting i jumps to the i-th code version. Note that this is performed for all memory barriers and not only for the first barrier after a safe-point. Thus, no analysis is required. This may seem wasteful, because we only need to branch at the first barrier(s) following a safe-point, but this waste will be eliminated later. In the specialized code, the barriers are not modified with any branches. They are set to the specialized barriers given by the developer.

The compiler then goes over the safe-points in all specialized paths and just before each safe-point, it installs an unconditional branch into the main code version (number 0), so that safe-points will only run on the main code version.



(a) The code fragment from Figure 1, specialized according to the naïve method. Two clones of the code are generated and each is specialized for a subset of the phases. A phase check is added after each safe-point and a branch to the adequate specialized code path follows.

(b) The code fragment from Figure 1 with our path specialization method applied. An unspecialized version runs code between the safe-point and the first barrier. A phase check transfer controls to the appropriate specialized code version before each barrier. A branch back to the unspecialized version is added before each safe-point. Part of the generated code is actually dead code that can be eliminated.

(c) The code fragment from Figure 2(b) after performing dead-code elimination. Dead code is depicted as opaque nodes and edges.

Figure 2. The code fragment from Figure 1 specialized using the various methods from Section 3.

The simple steps described above, are the only steps required to make the code work with path specialization. However, these steps will cause the creation of a significant amount of unreachable code. In particular, the original code is only used to jump into the cloned specialized paths prior to barriers after a safe-point, so the rest of its code is redundant. Eliminating dead code is a simple optimization that exists in most compilers and can be easily used to remove the non-relevant parts of this scheme. Thus, without any new required analysis, we get decently optimized code for path specialization.

This path specialization method is depicted in Figures 2(b) and 2(c). Figure 2(b) shows the cloning of the original code for $n = 2$. Two specialized code versions are created plus one unspecialized version. After each memory barrier in the unspecialized code, a phase check transfers control to one of the specialized code versions. Control returns to the unspecialized version unconditionally before any safe-point. Figure 2(c) depicts a clean version of the obtained code after dead code elimination has been performed. The dead code appears in this figure as semi-transparent nodes and edges. Note that much of the code is removed during dead code elimination.

In order to specify this algorithm more rigorously, we have to specify what a safe-point is. As mentioned earlier, a safe-point is a point in which the phase as viewed by a thread may change. For garbage collection, this usually means that the program thread is willing to cooperate with the garbage collector via some handshake mechanism, or by having all the relevant information ready for a collector inspection. The cooperation usually happens in one of two different approaches. In one approach, the program thread actively checks "once in a while" (i.e., in a safe-point) whether it needs to cooperate with the collector. The safe-point in this case is such runtime cooperation code that is executed between two program instructions. The other standard approach is to let the collector signal the program threads and halt them during execution. The threads may only be stopped at safe-points, where in this case, safe-points are points in between two program instructions that contain no specific code; what makes them safe-points is the fact that they are considered safe to stop the threads at. In the rest of this paper, we will think of a safe-point as a point between two program instructions, where cooperation may occur.

In path specialization, we have to pay special attention to method calls. When a method is called, a phase change may happen while the method code is executing, and so after returning from the method, we may need to change the code version that is being used. Thus, at the point after a method call, we want to make sure that we handle a phase change if one has occurred. To be on the safe side, we ensure that we are in the general/main code version just before calling the method. This means that we will also return to the general code version after returning from the method. In the main code version we will check the phase and jump to the appropriate code version prior to executing any barrier code. We will later relax this conservative approach by employing inter-procedural analysis and

actually check if a phase *may* change in a routine.[1] Similar consideration should be given to returning from native code, or when waking up from waiting on some event, since the phase might have changed since the last phase check. These cases can be handled in the same manner as method calls are handled and we do not explicitly discuss them further.

The code generation discussed above is specified in Algorithm 1. We assume that a program phase change must only be recognized at well-defined safe-points. We also assume that the designer has partitioned the phases into n subsets and has distilled the specialized barrier for each of the sets. Finally, we assume a test code that, given a phase number, tells which code version should be used, i.e., returns a number i, $1 \leq i \leq n$.

Algorithm 1 : Code Generation for Path Specialization

1. **Clone the code.** Clone the entire code n times, marking the original code as un-specialized, and each clone with an index i, $1 \leq i \leq n$. Maintain a mapping $Clone(i, j)$ which maps the pair (i, j) to the j-th instruction in the i-th clone. The un-specialized code is the 0 clone for this mapping.
2. **Partition basic blocks before a safe-point.** For each safe-point appearing after instruction j in any of the clones, split the basic block just after instruction j, and create a jump after instruction j into the un-specialized code at location $Clone(0, j+1)$. The point before the jump in the cloned code is not regarded as a safe-point anymore. The point after the jump and before executing $Clone(0, j+1)$ in the un-specialized code is regarded as a safe-point. (This may mean an actual runtime cooperation code or just declaring that it is safe for the collector to pause the thread execution.)
3. **Jump to the un-specialized code before calling a method.** Treat the point of execution that precedes a method call as a safe-point and apply the operation in the above step for each method call.
4. **Insert jumps towards specialized paths.** Go over all memory accesses in the code (that require a barrier). Suppose instruction j is such a memory access. For all code clones i, split the basic block just before instruction j. In the un-specialized code insert, just before instruction j, a phase selection test and a jump to the instruction $Clone(i, j)$, where i is the result of the selection test. In all clone codes keep the original flow of execution by inserting a jump from the first part of the basic block to the second.
5. **Remove dead code.** Run a control flow graph simplifier (or take special care) to remove all cloned code prior to the first memory access after a safe-point (or after method entry). Un-cloned code for which equivalent cloned code is always run (between a memory access and the following safe-point) should also get eliminated.

A Remark about Inlining. We assume that path specialization will be performed after the bulk of application code inlining. Method calls inlined after path specialization will result in a potentially unnecessary jump back to un-specialized code followed by an unnecessary phase selection test. The unnecessary jump back to un-specialized code is inserted by our algorithm prior to the soon-to-be-inlined method call; the unnecessary phase test will occur in the first memory accessing basic block of the soon-to-be-inlined callee.

[1] We assume that all methods have a single entry point and that method calls will always return to a specific code point (unless the method terminates due to a thrown exception).

4. Optimizations

Several further optimizations may be used to improve performance and reduce the code size of the code output by path specialization.

The techniques described in Section 3.2 above use dead-code elimination to avoid specialized analysis, yet, still be able to join code that follows a safe-point and precedes the first memory barrier. However, for a code path that starts at a barrier and ends at a safe-point and that does not contain any other barriers or safe-points, the same code sequences will appear in all n specialized versions of the original code (and will be identical to the code sequence in the original code). A more sophisticated algorithm for path specialization could perform tail merging for such code paths, in effect only creating the specialized code if doing so allows the use of specialized barriers relative to the original code. Doing so as part of path specialization requires some sophistication and some specialized analysis because the code is not necessarily linear and this part may contain several actual code paths. Alternatively, tail merging may be done separately as a general compiler optimization.

Various additional optimizations apply. To reduce code size growth, we may decide to not clone code that is infrequently executed. Not optimizing such code will not harm the execution, but not duplicating code may yield smaller code size. Similarly, we may decide not to clone code that has a small number of memory barriers. Another possible optimization is to let some of the work be done on the original code before cloning, thus, avoiding repeated work on the clones. For systems in which back-branches are safe-points, loop unrolling may be used to place safe-points further apart. An optimization that we have implemented is to have the basic block scheduler place block associated with slow paths separately, which has two consequences: first, fast path execution has better locality; and second, jumps to unspecialized code from fast path code are often elided, because the scheduler can juxtapose the blocks.

In what follows, we concentrate on inter-procedural optimizations. We stress that even without all these optimizations, path specialization yields dramatic improvements, as demonstrated in the measurement section below.

4.1 Inter-procedural Path Specialization

In Section 3, all method calls were considered potential safe-points. In particular, it followed that upon return from a method, control returned to the original code, or a check was made to determine which specialized path should be taken at this point. This is due to the conservative assumption that the called method (or one of the methods it calls, etc.) may contain a real safe-point, and then a phase change may occur, dictating a change in the selection of the specialized path *in all methods on the callstack*. Hence, in our conservative approach, when returning into a method, the phase selection must be rechecked. A code reachability analysis can be used to determine if any code reachable during the method call actually contains a real safe-point. If a real safe-point is not reachable during the call, the execution may proceed in the specialized path upon call to (and return from) the method execution, knowing that the collector phase has not changed. That can be simply done by calling the original method from the specialized path and then the return address will direct execution to the specialized path again. In this simple implementation, at entrance to the method, a check of the phase will still happen, directing the execution inside the method to the appropriate specialized version. But no check will be required at the exit.

To save the entry check as well and remain in a specialized version through the execution of the method (from which a safe-point is not reachable), that method itself may be specialized into n different methods. Each of the method versions handles a subset of the phases, rather than just having a single method body that contains various specialized versions of the code. The specialized versions of the code containing calls to such methods may then have the calls modified to directly call the appropriately specialized methods.

Alternatively, if the compilation environment supports methods with multiple entry points, a method may be specialized to have an original entry point plus up to n specialized entry points. Method calls in specialized code fragments may then be modified to call the appropriate entry point in the specialized method. This can be useful even when a safe-point does exist in the routine, avoiding the phase check on method entrance. Having multiple entry points is more advantageous than duplicating method codes because it allows some degree of code sharing. Note that even though the same code will be executed until the end, the return address will still direct execution to the calling path, which may be a specialized one.

Similarly, if the compilation environment supports multiple possible return addresses, then a method may be specialized to return to different code points depending upon which phase the collector is assumed be in at the end of the function.

The inter-procedural optimizations described above may be added as desired or needed. A modification of the safe-point specification using a reachability analysis, and specialization of entire methods, instead of just code inside the method, can be easily incorporated into the above algorithm. Using specialized entry and exit points is also straightforward.

5. Implementation

We implemented the path specialization mechanism in the Bartok compiler and runtime for C#, which has been developed at Microsoft Research. The Bartok compiler is an ahead-of-time compiler from CIL (or from C#) to native code.

The path specialization optimization has been implemented as a general optimization stage independent of the actual barrier used. The optimization stage assumes $n = 2$ (i.e., two specialized versions) and takes as arguments the specification of two barriers, each of which is used to insert barrier code for two disjoint subsets of garbage collector phases. Bartok operates at three IR levels: CIL [15], followed by a three-address code IR, and finally a low-level machine-dependent IR; path specialization is currently implemented within the three-address code IR. Path specialization makes the same safe-point assumptions as the Bartok runtime: namely, safe-points are at memory allocation and native calls, and any operation that leads to memory allocationor native calls. The specialized code is subjected to the compiler's general optimization framework, which includes dead code elimination. However, optimizations such as tail merging and predicate hoisting are not part of the optimization framework, so further code reduction is definitely possible. We have also not used any inter-procedural optimizations in our implemented version.

For each of the barriers we measured, two specialized barriers were implemented, where one handles the idle path, in which no barrier is required, and the other handles all the phases that are not idle. We have run measurements for each of the three types of barriers described below.

5.1 STOPLESS barriers

We first consider a heavy barrier used for an advanced real-time garbage collector. The STOPLESS collector is a recent concurrent real-time garbage collector that supports concurrent compaction, i.e., objects can be moved while the program threads concurrently are accessing or modifying them [23]. The collector employs read- and write-barriers for accesses to both reference and non-reference values. We denote this barrier by STOPLESS. We measure runs of the standard collector, and also a version that does not compact the heap and runs the garbage collector in the idle phase at all times. The latter is denoted STOPLESS nocopy. In the idle phase, the write-barrier described in Section 5.3 is used (as opposed to using any barriers). STOPLESS is idle most of the time; see [23] for detailed numbers of STOPLESS barrier activity.

5.2 Brooks barriers

Some incremental copying garbage collectors keep several versions of objects while moving them and they use forwarding pointers in each object to point at the latest version of the object. When accessing an object, it is costly to test whether an object has a forwarding pointer or not, thus, a self pointer is added to the latest version of each object. This way, following the forwarding pointer guarantees access to the latest object copy. A Brooks barrier employs an extra level of indirection on references. In the presence of null reference values for which it is not possible to use a forwarding pointer, a null check and branch may be necessary, but this test-and-branch code will only be used for reading of reference values rather than for all heap accesses. Dereferencing the forwarding pointers can be done using an *early-update* or a *lazy-update* strategy or using hybrids of the two.

The early-update strategy maintains the invariant that local and temporary variables are always updated with pointers to fresh copies. Pointers on the heap may contain references to stale copies of objects, but upon reading a reference from the heap, the program will dereference the forwarding pointer to obtain a reference to the representative memory block. The early-update strategy relies on the collector to update all pointers on the runtime stacks of mutator threads whenever objects are relocated. This is not a practical option for a concurrent collector that must update all thread stacks just after relocating each object.

Using the lazy-update strategy, local and temporary variables may contain references to non-representative memory blocks. Any read or write access of an object field using a reference may need to dereference the forwarding pointer before accessing the memory representing the field value. This option incurs a noticeable cost because all heap accesses are indirect.

A hybrid option is usually the preferred choice in this case. If forwarding pointers only change (or only have to be recognized as being changed) at known program points (e.g., garbage collector safe-points), then program analysis may be employed to determine that some references can only point to representative memory blocks and that dereferencing of the forwarding pointer is not necessary. The hybrid method is dereferencing a pointer when it is first accessed after a safe-point but not for subsequent accesses. Note the difference from path specialization, which branches once for all variables and does not need to perform an operation per variable. Thus, path specialization can improve also this clever use of a Brooks barrier, and it does, as will be shown in the measurements section below.

The lazy-update dereferencing strategy was originally described by Brooks [8], but all variants have subsequently been collectively described as Brooks barriers.

The Bartok runtime does not include a garbage collector that employs a Brooks barrier. However, the easy configurability of the runtime allowed us to add various kinds of Brooks barriers to a non-copying concurrent collector to allow us to measure the overhead of these barriers in the case where all forwarding pointers are self-pointers that point to the containing memory block. The barrier cost in an actual copying collector, for which some of the references are

not self-pointing, are likely to be higher due to cache locality issues. We denote by BROOKS the runs with a lazy-update barrier, and by BROOKS SUNK the runs with the hybrid version of the barrier. Note that because our collector never actually requires the Brooks barrier, we run *as if* the copying was in the idle phase all of the time.

5.3 A Concurrent Mark-Sweep style barriers

We also consider a write barrier of a concurrent mark-sweep collector similar to the collectors in [12, 11, 13, 14]. We denote the barrier employed in this collector the CMS (concurrent mark-sweep) barrier. The concurrent collector maintains a snapshot-at-the-beginning invariant and employs a write-barrier whose functionality changes according to the collector phases. In the idle phase the barrier does nothing. When the collector is active tracing the heap, the write-barrier is used to ensure that a potentially disconnected part of the object graph is not going to be missed by the collector. Except for a short time in the beginning of the collection, this barrier simply records referents whose pointing reference is modified, for later use by the collector. Namely, before a pointer is modified, the CMS barrier reads the memory word to be overwritten, and if it is a reference to an object that hasn't been previously noted by the collector, then the reference value is recorded in some mark-stack, or by marking the object's header, or both. In our implementation, the referent is added to a linked list using a CAS operation on a word in the object's header.

The CMS barrier is active in the trace phase but not in the sweep or idle phases of collection. See [23] for the percentage of time that our concurrent mark-sweep collector spends in each of these phases.

6. Measurements

To evaluate how path specialization affected the overhead of using various barrier implementations, we ran a set of experiments using the following barriers. First, we used the STOPLESS barriers in two configurations of STOPLESS: a default one and one that does not copy at all. Second, we checked two Brooks style barriers: the lazy-update version and the hybrid version. Finally, we checked the CMS barrier. We measured changes in execution time and code size. The test programs for this evaluation are shown in Table 1.

The JBB program had been translated from Java into C# as part of an unrelated project. The porting notes indicate that several scalable data structures have been replaced with non-scalable data structures, so the multiprocessor performance of this program should not be compared with that of the original Java program.

All measurements have been performed on an Intel Supermicro X7D88 dual x86 quad-core workstation running Microsoft Windows Server 2003 R2 Enterprise x64 Edition at 2.66GHz with 16GB RAM. For each program, each configuration was run in sequence, and the sequence was repeated a total of 5 times. The median execution time was chosen as representative. The standard deviation around the median was generally less than half a percent for the small programs, between 0.1 and 1.3 percent for sat, between 0.6 and 5.6 percent for lcsc, and between 0.3 and 3.8 percent for Bartok.

6.1 Throughput Measurements

Figure 3 illustrates the overheads of each of the barriers. CMS represent the write barrier of the concurrent mark-sweep collector. Brooks represents the lazy-update Brooks-style barrier, and Brooks sunk follows the forwarding pointers upon first use, and uses a data-flow analysis to propagate the knowledge that the reference has been forwarded in order to mostly avoid following a forwarding pointer more than once. STOPLESS runs the barriers in the default

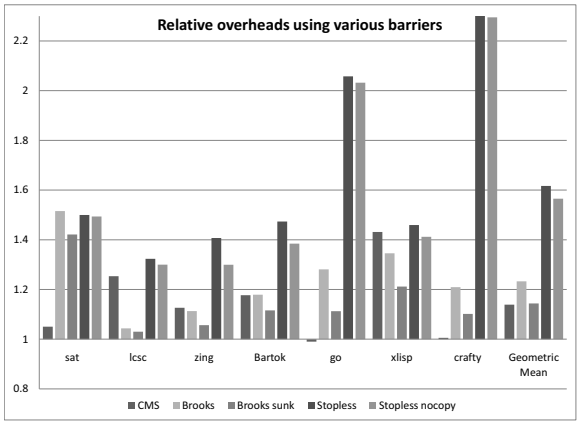


Figure 3. Overhead of barriers: execution times when using the barriers over execution times without executing the barriers.

collector configuration with object relocation, and STOPLESS no-copy represents the barriers when compaction is never executed, yielding a lighter barrier.

The overhead varies substantially between the benchmarks, because of the different frequencies of memory loads and stores. Looking at the geometric mean, the overheads range between 10-51%. These overheads are substantial and demonstrate the need for path specialization.

In order to measure barrier overheads, one must be able to turn them off during a run without crashing. For the Brooks-style and STOPLESS no-copy barriers, we just switched them on and off, without using a related collector. The unreachable objects were reclaimed with the concurrent mark-sweep (CMS) collector. For the STOPLESS barrier that does copy, we compared a run of the barrier and the partial compactor to a run with no barriers and no compaction. We believe that most of the overhead comes from the barriers, and therefore comparing the copying and non-copying runs shows mostly the overhead of the barriers. (Note that the results are not much different from those in which the collector does no copying at all.)

To measure the barriers of the CMS collector, we had to use a more intrusive method, because running without the barriers makes the collector miss objects during the trace, and the program later crashes. In order to correctly collect the objects, we ran the collector in three phases. First, the concurrent trace procedure marks objects as usual. Second, in an extra phase, the collector stops all threads and runs a stop-the-world tracing procedure. Finally, the concurrent sweep is performed. The second phase was introduced in order to ensure that the collector works correctly, even if the barriers are not executed. This allowed us to measure execution times with and without barriers in the first phase of the collector.

Next, we investigate the improvement in execution times when using path specialization. Figure 4 shows the gains from using path specialization. Higher is better, representing a higher ratio between the time to run the original version and the time to run the path specialization version. On average, the overall execution times were significantly reduced (around 30% reduction in execution times) for STOPLESS, and nicely reduced (a 2-6% reduction) for CMS and Brooks. For the Brooks barrier path specialization was effective on average, but sometimes caused a decrease in performance. We suspect that the branch prediction component might influence these fluctuations, since the Brooks barrier avoids conditional branches when possible. One can improve performance of path specialized

Benchmark	Types	Methods	Instructions	Objects Allocated	KB Allocated	Description
sat	24	260	19,332	8,161,270	171,764	SAT satisfiability program.
lcsc	1,268	6,080	403,976	8,202,479	426,729	A C# front end written in C#.
zing	155	1,088	23,356	12,889,118	928,609	A model-checking tool.
go	362	447	145,803	17,904,648	714,042	The commonly seen Go playing program.
xlisp	194	556	18,561	125,487,736	2,012,723	The commonly seen lisp implementation.
crafty	154	340	40,233	1,794,677	217,794	The Crafty chess program translated to C#.
Bartok	1,272	8,987	297,498	434,401,361	11,339,320	The Bartok compiler.
JBB	65	506	20,445	501,847,561	54,637,095	JBB ported to C#.

Table 1. Benchmark programs used for performance comparisons.

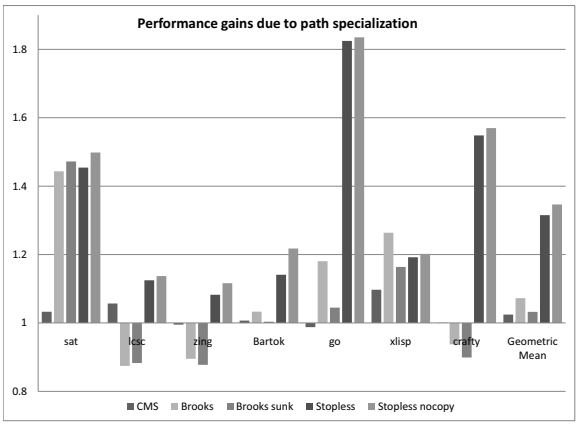


Figure 4. The performance gains achieved by using path specialization for the benchmark programs and the different barriers. Higher values mean higher gain and better performance when using path specialization.

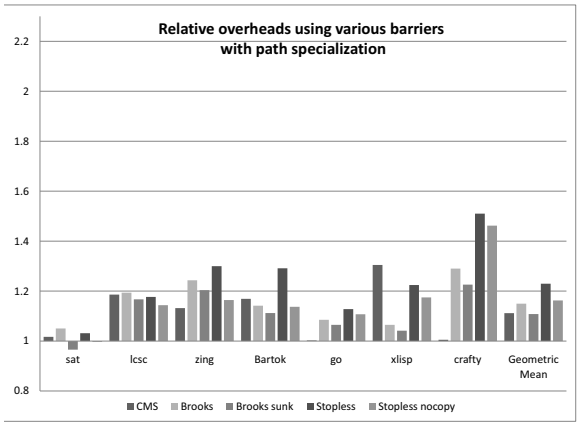


Figure 5. Relative execution times for the benchmark programs when using various barriers in conjunction with path specialization. For all but the CMS collector, the execution times have been normalized to the execution times of the programs when using a CMS-style barrier without path specialization. For the CMS collector, the execution times have been normalized against the no barriers case.

code by arranging the basic blocks appropriately; we have not attempted such optimizations.

The overheads of the barriers when path specialization is used are provided in Figure 5. The overheads are now between 7.5-18% on average. The main outliers have been tamed. For example, the crafty benchmark, whose STOPLESS overhead was 133%, is now running with an overhead of 50% only. The STOPLESS overhead of the go benchmark was reduced from 105% to 13%.

We note that the cost of the Brooks style barriers appears to be higher in our measurements than one should expect from reading Blackburn and Hosking's evaluation of barrier costs [6]. However, our measurements were performed on a machine with higher CPU to memory performance ratio, thus increasing the cost of memory operations, and using a compiler that does a better job of reducing other costs via optimizations, thereby increasing the relative cost of the barrier code.

We now move to the JBB program. This program runs several times with different number of warehouses (which equals the number of threads executing). We report similar measurements for the JBB program. Figure 6 illustrates the program performance for 1 to 6 warehouses when using the different kinds of barriers without using path specialization. A higher ratio means a higher overhead. Note that we did not include a check for CMS. The reason is that adding a stop-the-world phase to a program with many threads has a destructive effect on it, and in addition, we could not subtract the effects of this phase for a program that measures throughput rather than execution time. Looking at the other barriers, we see, again, that the overhead is substantial (between 15-45% on average) and optimizations are needed.

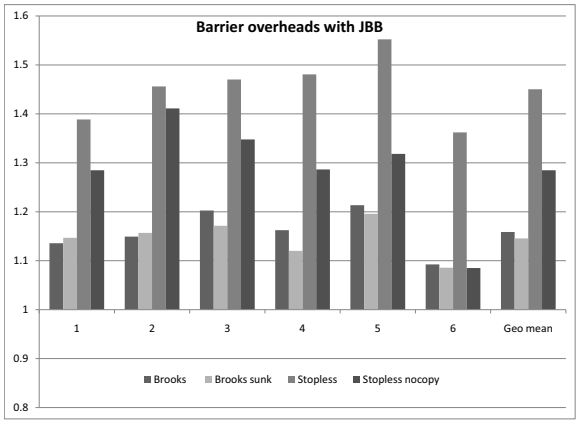


Figure 6. Throughput overhead of the various barriers without path specialization with the JBB program for various numbers of warehouses.

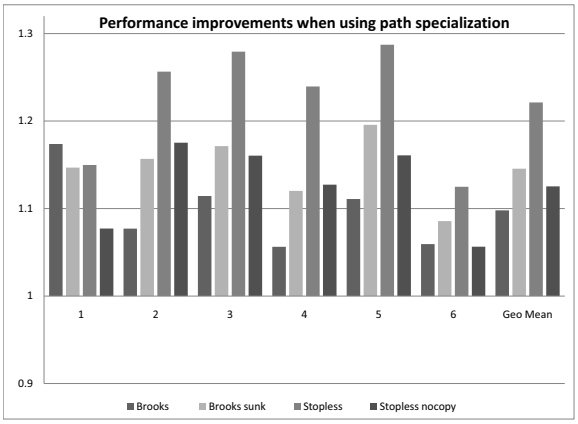


Figure 7. The relative performance gains due to path specialization measured with JBB for various warehouse numbers and the different types of barriers.

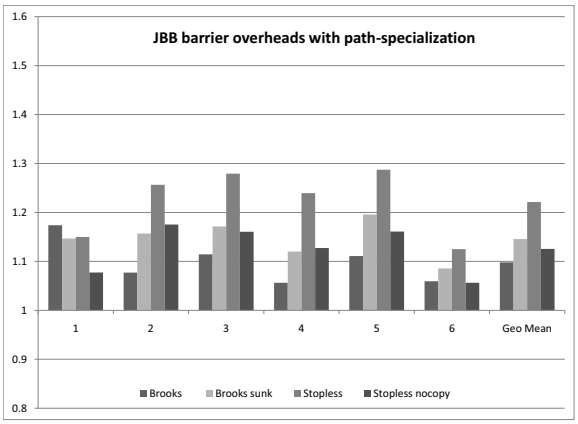


Figure 8. Throughput overhead of the various barriers with the JBB program when path specialization is used with various numbers of warehouses.

Figure 7 shows the gains obtained by adding path specialization. The average gains are between 1-19%, where the most substantial gains are for the highest overhead barrier, the STOPLESS barrier. Figure 8 provides the barrier overheads when path specialization is used.

6.2 Code Size

The duplication of code fragments required to insert specialized barrier operations may change the size of the code generated by the compiler. In general, the code duplication is expected to cause an increase in code size. Figure 9 shows how the use of path specialization changed the size of the generated code for the used barriers and benchmarks. (STOPLESS nocopy is not shown because it is equivalent to STOPLESS for this measurement). The bars show the ratio between the generated code with path specialization and without it. Values higher than 1 mean that the code has expanded when path specialization was applied.

For the two Brooks style barriers, the measurements indicate around 57% increase in code size, as one may expect. Surprisingly, for the CMS style barrier and the STOPLESS barriers, no substantial code growth is observed. This is due in large part to the compiler making different inlining decisions in the two compilation scenarios. Path specialization may increase the size of a methods, making the compiler not inline them. Path specialization in some cases also replaces multiple basic blocks on inlined barrier code with straight-line code for inlined specialized barriers. Other factors may also influence code size, such as register allocation, instruction scheduling, etc.

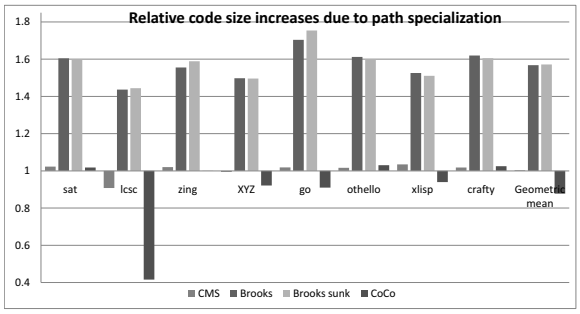


Figure 9. Relative size of generated code when using path specialization versus not using path specialization.

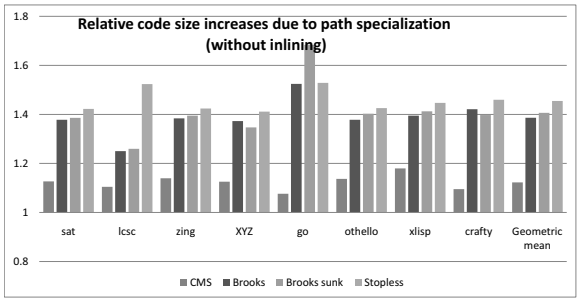


Figure 10. Relative size of generated code *without inlining* when using path specialization versus not using path specialization.

The decrease in code size for the lcsc benchmark when using the STOPLESS barriers is counter-intuitive and therefore we investigated it particularly. It is caused by the compiler choosing not to inline sequences of method calls used for object allocation when using path specialization. The benchmark is a compiler, which includes a parser. A substantial number of the parser methods include a switch statement on string tokens, which the C# compiler implements by means of hashtables from strings to branch indices. The hashtables are created and initialized upon first use. These methods thus contain an unusual number of object creation statements. The use of path specialization causes the compiler to choose not to inline the allocation methods due to a combination of the number of object allocation statements in the method and an increase in the size of the allocation code that was considered for inlining.

Although the code growth reported above is the one that will be seen in practice, we also wanted to check the actual growth when ignoring inlining effects. An illustration of the code growth while avoiding compiler inlining is shown in Figure 10. This time, the path specialization always has a cost. However, as can be seen, due to the optimization proposed in this paper and maybe other additional effects, the cost is not a doubling of the code (as it would likely have been using the naïve approach), but on average it reaches 40% even when a read-barrier is used.

Compilation overheads. For the STOPLESS barrier, path specialization causes a significant reduction in the number of basic blocks of methods due to eliminating phase checks. This has the inter-

esting effect that adding path specialization of this barrier actually leads to a reduction of around 20% in overall compilation time. Path specialization of the Brooks barriers leads to an increase in compilation time of around 30%, while path specialization of the CMS barriers often leads to a 10% decrease in compilation time.

7. Related Work

Using code duplication and specialization and transferring control back and forth between various specialized versions of the input code is a technique also used for bursty program tracing or sampling [1, 16]. The goal in that context is to perform program tracing once in a while at a minimal cost. For each program fragment, an original and an instrumented version is created. The original version is then modified to transfer control to the instrumented version when the code fragment has executed for a given number of times. The instrumented version is modified to transfer control back to the original version after performing an instrumented run of the fragment. Our work imports these ideas into the memory management world, making the required adaptations. A potential static analysis for reducing code-size was proposed by Arnold and Ryder [1] but it was not implemented or measured. In contrast, we propose a simple optimization that allows saving in code growth without the need for specialized analysis. This optimization was implemented and is reflected in our measurements. A similar method can be used for program sampling as well. Finally, we also propose some interprocedural optimizations that can be used in the memory management context.

Several papers have proposed to use static analysis in order to reduce the use of barriers. Nandivada and Detlefs have attempted to remove null checks using static analysis. Bacon and Vechev [26] propose an analysis that attempts to find redundant barriers, by checking the related lifetimes of objects. The success of such method depends on a good alias information for pointers on the heap. They have not implemented this method, but only checked its potential on traces, thus, it is difficult to know how effective it is. Joisha [19] proposes a detailed analysis that is targeted at reference counting collectors. His analysis extends the ideas in [26] substantially, but it only applies to root pointers, where information is easier to obtain reliably. This method is thus not applicable for typical concurrent, incremental, and real-time collectors, as they do not employ a barrier on root-pointer updates. Also, at this stage the analysis can only handle single threaded programs. Our method is much simpler and can work with advanced multithreaded programs and collectors. The Metronome collector [3] reduces the cost of the Brooks read-barrier for incremental collectors by eagerly updating root pointers and avoiding repeated de-referencing when no safepoint exists in the code.

8. Conclusion

Garbage collectors are becoming more popular in modern programming languages, and many of them employ memory barriers. In this paper, we have presented *path specialization*: a new optimization method for reducing the costs of memory barriers for memory managers that use phases. Path specialization was shown effective for a variety of garbage collectors including incremental, concurrent, and real-time collectors. A naïve implementation would imply a large code increase, or an introduction of a specialized static analysis to reduce code duplication. We have presented a simple method for obtaining much of the possible advantages relying only on dead-code elimination, which is available in most compilers. The resulting method is easy to implement and we have implemented it upon the Bartok compiler and runtime system. Measurements with various collector barriers show drastic reductions in the overhead of *read*-barriers and nice reductions in the overhead of *write*-barriers.

References

[1] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *PLDI*, 2001.

[2] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding view. In *OOPSLA*, 2003.

[3] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *POPL*, 2003.

[4] Henry G. Baker. List processing in real-time on a serial computer. *CACM*, 21(4):280–94, 1978.

[5] Katherine Barabash, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, Yoav Ossia, Avi Owshanko, and Erez Petrank. A parallel, incremental, mostly concurrent garbage collector for servers. *TOPLAS*, 27(6):1097–1146, November 2005.

[6] Stephen M. Blackburn and Tony Hosking. Barriers: Friend or foe? In *ISMM* 2004.

[7] Hans-Juergen Boehm, Alan J. Demers, and Scott Shenker. Mostly parallel garbage collection. *SIGPLAN Notices*, 26(6):157–164, 1991.

[8] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. *LFP* 1984.

[9] Perry Cheng and Guy Blelloch. A parallel, real-time garbage collector. In *PLDI* 2001.

[10] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *CACM*, 21(11):965–975, November 1978.

[11] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL* 1994.

[12] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ML. In *POPL* 1993.

[13] Tamar Domani, Elliot Kolodner, and Erez Petrank. A generational on-the-fly garbage collector for Java. In *PLDI* 2000.

[14] Tamar Domani, Elliot K. Kolodner, Ethan Lewis, Elliot E. Salant, Katherine Barabash, Itai Lahan, Erez Petrank, Igor Yanover, and Yossi Levanoni. Implementing an on-the-fly garbage collector for Java. In *ISMM* 2000.

[15] ECMA. Standard ECMA-335, Common Language Infrastructure (CLI). 2006.

[16] Martin Hirzel and Trishul Chilimbi. Bursty tracing: A framework for low-overhead temporal profiling. In *FDDO* 2001.

[17] Richard L. Hudson and J. Eliot B. Moss. Sapphire: Copying GC without stopping the world. *Joint Java Grande — ISCOPE* , 2001.

[18] Galen C. Hunt and James R. Larus. Singularity: Rethinking the Software Stack. In *Operating Systems Review*, 2007.

[19] Pramod Joisha. Compiler optimizations for nondeferred reference-counting garbage collection. In *ISMM* 2006.

[20] H. T. Kung and S. W. Song. An efficient parallel garbage collection system and its correctness proof. In *FOCS*, 1977.

[21] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. *TOPLAS*, 28(1), January 2006.

[22] David A. Moon. Garbage collection in a large LISP system. *LFP* 1984.

[23] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgaard. Stopless: A real-time garbage collector for modern platforms. In *ISMM* 2007.

[24] Tony Printezis and David Detlefs. A generational mostly-concurrent garbage collector. In *ISMM* 2000.

[25] Guy L. Steele. Multiprocessing compactifying garbage collection. *CACM*, 18(9):495–508, September 1975.

[26] Martin Vechev and David F. Bacon. Write barrier elision for concurrent garbage collectors. In *ISMM* 2004.