

Write Barrier Elision for Concurrent Garbage Collectors

Martin T. Vechev
Computer Laboratory
Cambridge University
Cambridge CB3 0FD, U.K.
mv270@cl.cam.ac.uk

David F. Bacon
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, U.S.A.
dfb@watson.ibm.com

ABSTRACT

Concurrent garbage collectors require write barriers to preserve consistency, but these barriers impose significant direct and indirect costs. While there has been a lot of work on optimizing write barriers, we present the first study of their elision in a concurrent collector. We show conditions under which write barriers are redundant, and describe how these conditions can be applied to both incremental update or snapshot-at-the-beginning barriers. We then evaluate the potential for write barrier elimination with a trace-based limit study, which shows that a significant percentage of write barriers are redundant. On average, 54% of incremental barriers and 83% of snapshot barriers are unnecessary.

General Terms

Experimentation, Languages, Measurement, Performance

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features—*Dynamic storage management*; D.3.4 [Programming Languages]: Processors—*Memory management (garbage collection)*; D.4.2 [Operating Systems]: Storage Management—*Allocation/deallocation strategies, Garbage collection*

1. INTRODUCTION

Garbage collection is a useful feature of most modern object oriented and functional programming languages. Garbage collection reduces memory management errors that occur in languages with explicit heap management. Ideally, a collector would reclaim just enough memory, induce no application pause times, and maximize throughput. In reality, there exist many different collector implementations each with its own advantages and disadvantages.

In this paper we focus on concurrent garbage collectors (also called “on-the-fly” collectors). The main advantage of concurrent over stop-the-world garbage collectors is the reduction of application pause times. A concurrent collector usually runs in its own thread, either on the same processor as the application threads or in

parallel with them on another CPU. There exist a number of different implementations for concurrent or mostly-concurrent collectors [3, 12, 18, 21]. In contrast to stop-the-world collectors, application threads (mutators) are not stopped or are stopped for a short quantum. Achieving even and short application response times comes at the price of decreased throughput. Therefore the goal for concurrent collectors is keeping their pause times low, while simultaneously increasing the application throughput.

In order to prevent the erroneous reclamation of reachable objects, the runtime system utilizes a synchronization technique called a *write barrier*. A write barrier traps pointer stores to the heap and records some information that prevents race conditions between the mutator and the collector. It performs a different procedure depending on the granularity of the barrier. For an excellent survey of barrier techniques, see [20].

Write barriers in concurrent collectors are undesirable for a number of reasons:

- Direct run-time cost of executing the additional instructions at every pointer store;
- Indirect run-time cost due to I-cache pollution by write barrier code and suppression of compiler optimizations across the barrier, which is generally a collector safe point;
- Collector cost to process the information recorded by the barrier, and in the case of incremental update barriers, the cost of re-tracing some portion of the heap;
- Space cost for the write barrier data structures (usually sequential store buffers);
- Reduction in minimum mutator utilization (MMU) [4, 9];
- Increased compile time; and
- Increased application code size.

While there has been work on reducing the cost of generational write barriers [7], the costs for concurrent write barriers is generally significantly higher, so those results often do not apply.

We are particularly concerned with the additional work that write barriers create for the garbage collector. The write barrier remembers pointers which the collector must process at the end of its collection cycle. The collector must process all pointers stored by the write barrier, even though processing only a fraction of them may suffice to preserve application correctness. Reducing the number of pointers the collector must process can indirectly lead to increases in mutator utilization: because the collector will have less work to perform, more processing time can be dedicated to application activity.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISMM'04, October 24–25, 2004, Vancouver, British Columbia, Canada.
Copyright © 2004 ACM 1-58113-945-4/04/0010 \$5.00.

The focus of this paper is removing unnecessary write barriers in concurrent garbage collectors. In concurrent schemes, write barriers protect against interleaving between the collector and the mutators. Such interleaving can cause reachable objects to be collected incorrectly. Write barriers for concurrent schemes can be broadly classified as incremental-update and snapshot-at-the-beginning; see [27] for a complete discussion. Steele [25] and Dijkstra [11] pioneered incremental write barrier techniques, while snapshot barriers became popular with Yuasa [28].

In this paper we present elimination conditions for removing redundant write barriers. We consider the three main write barrier types. Although there are many variations of Dijkstra, Steele and Yuasa, they all can be reasoned about using our conditions.

We perform a trace-based limit study that provides an upper bound on how many write barriers can be removed for simple elimination conditions. Such information suggests which elimination conditions are most useful to concentrate on during the static analysis. Our results indicate that write barriers are unnecessary in the majority of cases. If all elimination conditions are utilized, on average 54% of Dijkstra/Steele and 83% of Yuasa write barriers are redundant.

We also perform various forms of correlation analysis to study how barrier elimination relates to object attributes such as size, lifetime, and popularity.

The dynamic analysis is based on traces generated by an instrumented Java virtual machine (Jikes RVM 2.2.0 [2]). We consider benchmarks from the SPECjvm98 suite as well as other Java benchmarks.

The paper is organized as follows: Section 2 presents background on write barriers. Section 3 presents the write barrier elimination conditions. In Section 4 we discuss our results from the dynamic trace analysis. Section 5 discusses related work and finally we conclude and present our ideas for future research in section 6.

2. BACKGROUND

In this section we introduce some necessary terminology. We discuss why write barriers are necessary for correctness and how they are utilized in some of the more recently introduced on-the-fly collectors.

A garbage collection cycle usually consists of three main phases: root set marking, tracing through the heap connectivity graph starting from the objects in the root set, and finally reclaiming the space of all unreachable objects. The root set marking phase typically consists of marking the objects reachable from thread stack frames and CPU registers. In the case of Java, root set pointers also contain JNI global references and class class static data. Depending on the implementation, other pointers could also be categorized as root set pointers.

In the second phase, the collector traces through the heap and marks as live all objects transitively reachable from the root set. In the last phase, all objects which have not been marked as live are reclaimed as garbage. In Dijkstra’s tri-color terminology, if an object is white, then it is either garbage or has not been reached by the collector yet. If it is gray, then it has been marked as reachable for this collection cycle but its contents have not been scanned yet. Once all the pointers an object contains are marked as reachable (gray), the color of the object turns from gray to black.

In a concurrent collector, the mutators can preempt the collector and change the heap connectivity graph during the collection cycle. The *write barrier* is a synchronization mechanism which protects the collector from reclaiming reachable objects. To see why write barriers are necessary, consider the sequence in Figure 1 and let us suppose we operate without applying write barriers. Initially

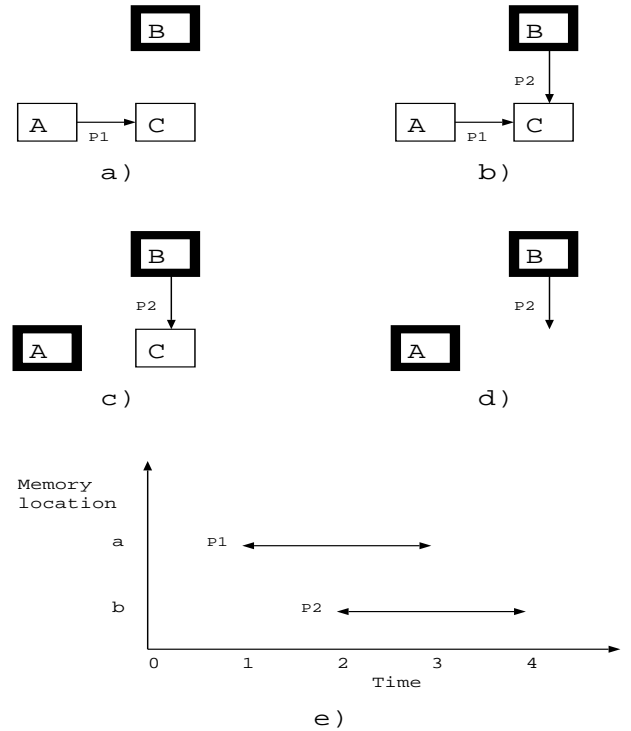


Figure 1: A sequence of pointer operations that can lead to erroneous collection of the live object C.

we have a part of the connectivity graph as shown in Figure 1(a). The collector is in the middle of its second (tracing) phase and has already scanned object B (that is, B is black). Object A, which contains pointer p_1 to object C, is reachable but has not yet been reached by the collector.

In the next step, shown in Figure 1(b), an application thread (mutator) introduces pointer p_2 to object C. The pointer is introduced in a part of the heap which the collector has already scanned, that is, in object B. Object A is still not marked.

In Figure 1(c), the mutator destroys pointer p_1 and the collector then scans object A, not finding pointer p_1 there. The collector then proceeds to its sweeping phase with object C unmarked. It collects object C incorrectly, as shown in Figure 1(d), making p_2 a dangling pointer.

In terms of pointer lifetimes, we observe that pointer p_1 is created before pointer p_2 and is deleted while p_2 is still active. We denote the start of a pointer’s lifetime by $S(p)$ and its end by $E(p)$. Thus $S(p_1) < S(p_2)$, $S(p_2) < E(p_1)$ and $E(p_1) < E(p_2)$. We observe that if we do not utilize write barriers, then at time 3 in Figure 1(e), the collector may reclaim object C although it is still reachable through pointer p_2 .

The situation depicted in Figure 1(e) is precisely the pointer lifetime combination against which write barriers are meant to protect. Nonetheless, as discussed in the following sections, we can eliminate redundant write barriers if we have a different combination of pointer lifetimes and/or if we have further information on the types of pointers involved in the combination.

Three approaches are commonly used to protect object C:

Steele’s method: When pointer p_2 is installed, we mark object B as live, that is, the object to be rescanned. This action undoes the work of the collector. This is an incremental technique.

```

SteeleWB(source, slot, newtarget)
{
  if (tracingOn && newtarget != NULL)
    store(source);
}

```

(a) Steele Write Barrier

```

DijkstraWB(source, slot, newtarget)
{
  if (tracingOn && newtarget != NULL)
    store(newtarget);
}

```

(b) Dijkstra Write Barrier

```

YuasaWB(source, slot, newtarget)
{
  oldtarget = source[slot];
  if (tracingOn && oldtarget != NULL)
    store(oldtarget);
}

```

(c) Yuasa Write Barrier

Figure 2: Write barrier pseudo-code. Steele’s and Dijkstra’s barriers support an incremental update collector, while Yuasa’s supports a snapshot-based collector.

Dijkstra’s method: Upon installing pointer p_2 in object B we mark C as reachable. That is, we prevent the introduction of a pointer that resides in a scanned part of the heap to a part of the heap that has not yet been scanned by the collector. This is also an incremental mechanism.

Yuasa’s method: Upon overwriting pointer p_1 in Figure 1(c), we mark object C as live. This is a snapshot technique which does not allow the destruction of any path in the heap connectivity graph.

Wilson [27] has classified concurrent write barrier techniques as either incremental-update (the Dijkstra and Steele barriers) or snapshot-at-the-beginning (the Yuasa barrier). We use this terminology throughout.

Each technique has its advantages and disadvantages. From the perspective of reducing floating garbage, Steele’s method is least conservative, while Yuasa’s will not collect garbage that has become such during the current collection cycle. A problem with Steele’s method is that it increases collector work. This is because, we are graying the source object where the pointer is stored into. In order to uncover the stored pointer, the collector would need to rescan all the slots of the source object, even if many of the slots are NULL or have already been marked.

On the other hand, a Yuasa-style barrier will usually mark fewer objects than Dijkstra since Yuasa barriers are active only when a pointer value is overwritten, while Dijkstra always marks the object pointed to by the stored reference. Steele’s and Dijkstra’s write barriers protect the same object, which is the object whose address is written into a memory location, while Yuasa’s barrier protects the object denoted by the overwritten pointer. A significant disadvantage of the Yuasa barrier is that it must load the old value of the pointer.

Several variations of Yuasa’s and Steele’s style write barriers exist. With Yuasa, instead of marking object C when pointer p_1

is overwritten, we could mark all objects that object A points to. Such an approach is utilized by Azatchi et al. [3]. In contrast, Domani and Kolodner [12] takes the usual approach and marks object C only. There are trade-offs in each approach. While Azatchi et al. claim increased application locality, they do so at the expense of holding a special thread-local pointer inside each object. The traditional approach of marking only one object has the disadvantage that in the current collector cycle we could possibly mark more objects than necessary if the same slot is mutated more than once. With Azatchi et al.’s method this problem is alleviated since we know we have marked all the pointers in an object. Steele’s barrier is widely used with various granularities. One could mark object B, but one could also mark a whole card for rescanning, as in Barabash et al. [5].

Regardless of the granularity and the type of the write barrier, its purpose is protecting object C from being collected erroneously.

The write barrier itself typically consists of two parts, as shown in Figure 2: a *filter* which checks whether a pointer needs to be stored and a *slow path* which performs some action. The slow path usually stores a pointer into a log buffer or marks an object as reachable. Depending on the write barrier type, the two parts of the barrier may differ slightly.

Generally, there exists a trade off between how precise the write barrier is and how much work it creates for the collector. In regard to write barrier execution speed, Steele barriers have the potential to be the fastest of the three. A Steele write barrier can omit the filter check and simply color the source object as reachable. Dijkstra and Yuasa barriers need to perform the filter check. A Dijkstra barrier needs at least a comparison instruction to check whether the new target is NULL. A Yuasa barrier needs a load and a comparison instructions to check whether the overwritten value is non-NULL. See Detlefs et al. [10] for a discussion on the trade off.

A *redundant write barrier* is a barrier which when removed will not cause the reclamation of a reachable object, regardless of the scheduling between the mutator and the collector.

Two complementary approaches exist for eliminating write barriers statically. The first approach proves that the filter always evaluates to false. That is, either the pointer in question is always null or the collector is not in its tracing phase. Clearly, we cannot evaluate statically whether the collector is tracing. Therefore, the only plausible condition left is statically analyzing for the NULL check. In the case of Yuasa, if the overwritten pointer is null [19] we do not need a write barrier. Similarly, for Dijkstra and Steele, if the new pointer is null we can eliminate the write barrier.

The second approach is possible if we are not able to statically determine the filter evaluates to false, but we are able to show that the slow path is redundant. In such cases we can also remove the write barrier. Elimination conditions for proving the redundancy of the slow path are significantly more complex and are described in the next section.

Nonetheless, the two approaches are complementary and for maximum efficiency should be utilized together.

One possible implementation could use a Steele-like fast write barrier which ignores the filter check and simply colors the source object gray. To reduce the amount of work the collector must do, we could combine the static approach which will utilize the elimination conditions in this paper together with the dynamic solution presented in Detlefs et al. [10].

3. ELIMINATION CONDITIONS

In this section we present four conditions for eliminating redundant write barriers for non-null pointers. That is, these are conditions that deal with detecting redundancy in the slow path of

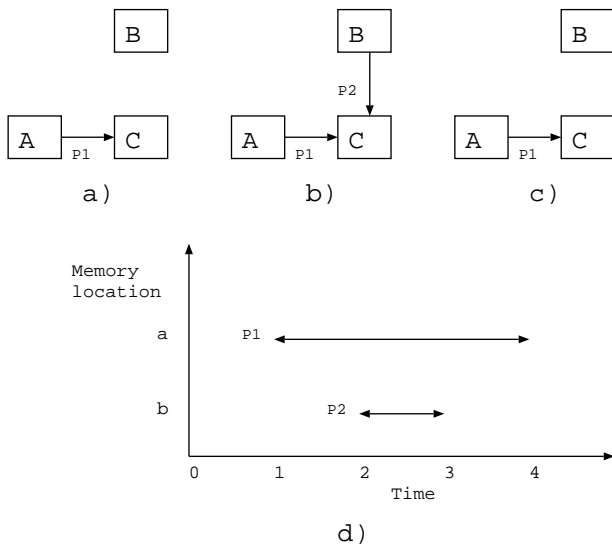


Figure 3: Single Covering Condition (SCC)

the write barrier. Each condition by itself is sufficient for eliding a write barrier. The conditions are categorized into two distinct elimination classes: *covering conditions* and *allocation conditions*. Covering conditions are more general in the sense that they apply to all pointers (heap, stack, global). In addition, covering conditions apply to both incremental update (Dijkstra/Steele) and snapshot (Yuasa) write barriers, while allocation conditions are valid only for incremental barrier elimination. Regardless of the elimination class, the fundamental property behind the reasoning for write barrier removal is the lifetimes of the pointers to the particular object in question.

Reasoning about Dijkstra and Steele barriers is essentially the same because they fundamentally protect the same object, namely the one pointed to by the newly installed pointer.

3.1 Covering Conditions

The main observation behind covering elimination conditions is that write barriers performed on pointers which do not determine the lifetime of an object are not required. For example, if an object is being pointed to by a long-lived static pointer and during this period, a number of write barriers protecting the object are executed, then we could reason about their elimination. In the next two sections, we present two covering elimination conditions. The first condition is more intuitive and deals with cases when the lifetime of one pointer allows the barrier on another one to be eliminated. The second condition extends these ideas by using multiple pointers with write barriers to act as single “virtual” pointers with a longer cumulative lifetime.

3.1.1 Single Covering Condition

We first present an elimination condition that deals with barrier elimination in a pair-wise fashion, that is, it only considers combinations of two pointers. The condition is illustrated in Figure 3.

In Figure 3(a), pointer p_1 exists from object A to object C and object B contains no (non-null) pointers. Next, in Figure 3(b) the application installs a pointer p_2 from object B to object C. If an incremental update (Dijkstra or Steele) barrier is used, then we must perform a write barrier when pointer p_2 is installed in object B. The application proceeds and in Figure 3(c), it overwrites pointer p_2 with another value. If we are using a snapshot (Yuasa) mechanism,

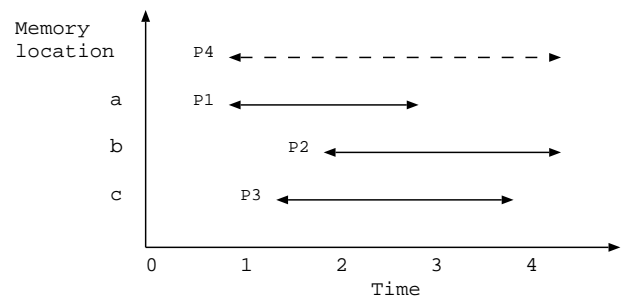


Figure 4: Multiple Covering Condition (MCC)

then we would need to apply a write barrier at the time pointer p_2 is overwritten.

We observe that throughout these three execution steps, the lifetime of pointer p_1 covers the lifetime of pointer p_2 , that is, pointer p_1 was installed before p_2 and p_2 ended before p_1 . The timing relationship between pointers p_1 and p_2 is illustrated in Figure 3(d). Since the lifetime of p_2 is entirely contained within the lifetime of p_1 , we call p_1 a *covering pointer* with respect to p_2 .

We observe that the write barrier performed on pointer p_2 is not necessary, regardless of whether it is an incremental update barrier which occurs at the beginning of the pointer’s lifetime, or a snapshot barrier which occurs at the end of the pointer’s lifetime. Regardless of the particular scheduling between the mutators and the collector, pointer p_2 does not determine the lifetime of the object. Therefore protecting the object from being collected by applying write barriers through pointer p_2 is not needed. Stated formally:

SINGLE COVERING CONDITION (SCC): *If pointers p_1 and p_2 point to object C and if $S(p_1) < S(p_2)$ and $E(p_1) > E(p_2)$, then the write barrier for pointer p_2 can be safely eliminated. For an incremental update barrier this eliminates the barrier when p_2 is stored, and for a snapshot barrier this eliminates the barrier when p_2 is overwritten.*

If we consider a set S of pointers to the object at time t , a partial order relation can be formed between the elements in the set S . A pair (p_1, p_2) of pointers belongs to the partially ordered set if and only if p_1 covers p_2 as defined in the single covering condition. An interesting item for future work is determining how easy would it be to modify the program so that every pair of elements in the set S has a least upper bound. That would be a pointer p_x which would cover all other pointers. Note that if we introduce such a pointer p_x , we would not require any write barriers on pointers to that object. In practice, it would be difficult to introduce p_x precisely, because that would be equivalent to knowing exactly when the object dies. Therefore, p_x could be introduced with an extended lifetime. The length of pointer p_x implies a question of trade-off between keeping the object live for a longer period than necessary versus the overhead of the write barrier.

3.1.2 Multiple Covering Condition

In this section we generalize the single covering condition. In the previous section, we considered the case where one pointer covers another pointer and hence we can eliminate barriers on the covered pointer. The material in this section extends the idea of coverage to multiple pointers. If we are presented with three pointers, could we still eliminate a write barrier provided that the single covering condition cannot? The condition presented in this section is a generalization of the single covering condition to three or more pointers.

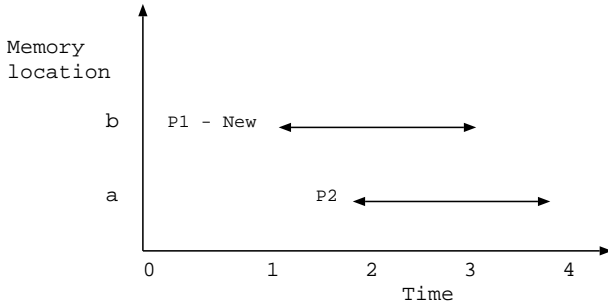


Figure 5: Single Allocation Condition (SAC)

The elimination condition is illustrated in Figure 4: we observe that the single covering condition does not hold for any combination of two pointers. For example, we cannot establish a covering relation between (p_1, p_2) , (p_2, p_3) or (p_1, p_3) . Therefore, when presented with such an example, we cannot eliminate any write barriers with the single covering condition.

However, we can generalize the single covering condition. While a pointer may not be covered by a single pointer, there may be multiple pointers that together cover it. Once again, if a write barrier is performed on a pointer which does not determine the lifetime of the object, then the write barrier is redundant. In Figure 4, we observe that pointer p_3 is covered from one side (start time) by pointer p_1 and from the other side (end time) by pointer p_2 . Additionally, p_2 starts when p_1 is active. Therefore, pointer p_3 is covered jointly by pointers p_1 and p_2 and we claim that we do not require a write barrier on pointer p_3 .

We observe that if pointers p_1 and p_2 are joined together, we obtain a new virtual pointer p_4 and now we can apply the single covering condition of the previous section to the combination (p_4, p_3) . Note that the write barrier at the end of p_1 (for snapshot barriers) or the beginning of p_2 may *not* be eliminated if the combined pointer is used as the basis for elimination.

The condition generalizes to an arbitrary number of pointers. We simply join the overlapping times of the pointers of interest. Stated formally,

MULTIPLE COVERING CONDITION (MCC): *Let p_1, p_2, \dots, p_n and q be pointers to an object C , such that $S(p_{i+1}) > S(p_i)$ and $S(p_{i+1}) < E(p_i)$ and $S(p_1) < S(q)$ and $E(q) < E(p_n)$. Assuming that none of the write barriers on p_i are eliminated, then the write barrier on q can be safely eliminated.*

The single covering condition in the previous section is a special case of the multiple covering condition. Essentially, virtual pointer p_4 in Figure 4 corresponds to a physical pointer in the single covering condition. As with the single covering condition, the multiple covering condition will hold for any interleaving between the mutators and the collector and applies to both incremental update and snapshot write barriers.

Maximal elimination will be achieved by finding the smallest set of pointers p_i covering a given interval, and then eliminating the barriers which they cover. However, this is largely of theoretical interest, since in practice it will be difficult to apply the multiple covering condition statically, and there are unlikely to be multiple choices about which pointers to select to create a covering.

3.2 Allocation Conditions

In this section we show how knowledge about the creation of an object can be used to eliminate additional barriers. Unlike the

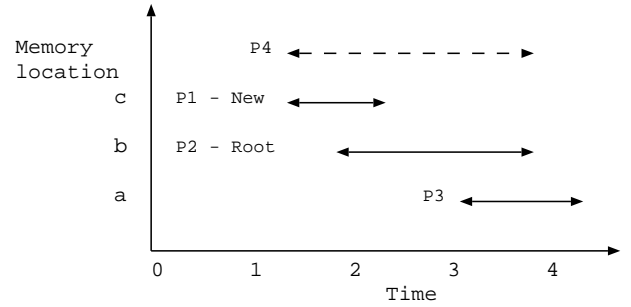


Figure 6: Multiple Allocation Condition (MAC)

covering conditions, these *allocation conditions* only apply to incremental update (Dijkstra/Steele) barriers.

3.2.1 Single Allocation Condition

Consider the pointer lifetimes shown in Figure 5. Pointer p_1 is the result of allocating the object in question. The resulting pointer is placed in stack/register location b . Pointer p_2 is a pointer from a heap location a .

Assume that the collector (a) marks all roots before commencing the heap marking phase, and (b) allocates new objects as reachable (black or gray). Note that many incremental update collectors allocate objects as unreachable (white) and rescan the stack(s) to ensure that such objects are not lost, so this technique may not always be applicable. At any rate, if (a) and (b) are true then an incremental update barrier on pointer p_2 can be eliminated.

At the time the write barrier is to occur, that is, at time 2 in Figure 5, the object will already be marked as reachable (black or gray). There are two cases to consider. First, if pointer p_1 was introduced while the collector had already commenced its tracing phase, then the object would have been created reachable (black or gray) already. On the other hand, if p_1 was introduced before the collection started, then the corresponding object would have been marked as reachable in the root scanning phase. That is, at the time pointer p_2 is introduced, the object is already guaranteed to be marked as reachable.

The allocation condition relies on the fact that objects allocated reachable (black) include an implicit barrier, and therefore it is not possible for the mutators to “hide” the pointer to the object from the collector by moving it between different memory locations. More formally the condition can be stated as follows:

SINGLE ALLOCATION CONDITION (SAC): *In a collector that allocates objects as marked, if p_1 and p_2 are pointers to object C and pointer p_1 is introduced as a result of creating object C (p_1 is a root pointer) and $S(p_1) < S(p_2)$ and $S(p_2) < E(p_1)$, then the incremental update (Dijkstra or Steele) write barrier on pointer p_2 can be eliminated.*

That is, if pointer p_2 is created within the lifetime of pointer p_1 and p_1 is the result of allocating the object, then incremental update barriers on p_2 are unnecessary.

There is no corresponding allocation condition for snapshot barriers. At the time an incremental update barrier happens, we guarantee that the object is already marked as reachable. We cannot make the same guarantee at the time a Yuasa barrier occurs. The collection cycle could have started just after the pointer to the allocated object ended its lifetime, that is, just after p_1 ends in Figure 5 (i.e. at time 3.5). Then the object would not have been allocated as marked at that time and might be collected incorrectly.

Program	Description	Input	Objects Allocated	Bytes Allocated
power	Solves the power system optimization problem	-	783,406	73,228,028
deltablue	Constraint solver	-	443,923	77,502,904
bh	Solves the N-body problem using hierarchical methods	-b 500 s 10	639,675	83,503,920
health	Simulates Columbian health care system	-1 5 t 500 s 1	1,196,725	86,718,316
ipsixql	Performs queries against persistent XML document	3 2	5,941,536	351,117,828
xalan	XSLT tree transformation language processor	3 2	4,420,252	488,960,484
SPECjvm98				
compress	Modified Lempel-Ziv method (LZW)	-s100	2204	159,951,240
mpegaudio	Decompresses ISO MPEG Layer-3 audio files	-s100	2096	51,372,816
db	Performs database functions on memory resident database	-s100	3,213,006	133,782,036
javac	Java compiler from JDK 1.0.2	-s100	6,376,872	579,746,244
mtrt	Multi-threaded raytracer	-s100	4,594,918	237,305,784
jack	A Java parser generator	-s100	7,471,385	473,120,964

Table 1: Benchmarks

3.2.2 Multiple Allocation Condition

The condition presented in this section extends the single allocation condition and similarly applies to incremental update write barriers only. As with the multiple covering condition, we can view several overlapping pointers as comprising a single virtual pointer which then allow another pointer to be eliminated using the reasoning of the single allocation condition.

The condition is depicted in Figure 6. Assume that p_1 is the result of allocating an object and that p_2 is another root pointer and p_3 is a heap pointer. We cannot apply the single allocation condition on the pair (p_1, p_3) because p_1 and p_3 have disjoint lifetimes and we cannot apply the single allocation condition to (p_2, p_3) because p_2 is not a pointer resulting from object allocation. Nonetheless, we observe that the start of p_3 is covered by p_2 , whose start in turn is covered by p_1 . Subsequently, the start of p_3 is covered jointly by p_1 and p_2 .

Intuitively, since the roots (stack pointers) are treated as a single unit, it does not matter if a pointer is copied from one stack location to another and then the first pointer is overwritten, because the object is always being pointed to from somewhere in the stack.

Similarly to the multiple covering condition, we reason about joining pointer lifetimes together. That is, p_1 's and p_2 's lifetime are joined to create a virtual pointer p_4 . The single allocation condition of the previous section can now be applied between (p_3, p_4) .

The main difference between the multiple covering condition and the multiple allocation condition presented in this section is a restriction on the type of pointers that could be joined. The restriction states that the lifetime of virtual pointer can only be composed of root pointers. That is, if p_2 was not a root pointer, then we would not have been able to apply this condition.

To see why the multiple allocation condition can not be applied when the chain of pointers are not all root (stack) pointers, consider the case in Figure 6 if p_2 were a heap pointer. If collection begins after p_1 is popped off the program's stack, and before p_3 is created, then the collector will not see p_1 when it scans the stack and may not see p_2 if p_2 is overwritten before the collector scans it.

Stated formally, the elimination condition is

MULTIPLE ALLOCATION CONDITION (MAC): Let p_0, \dots, p_n be root pointers to an object C and q be a heap pointer to object C , such that $S(p_{i+1}) > S(p_i)$ and $S(p_{i+1}) < E(p_i)$ and p_0 is the pointer returned by the creation of object C . Then if $S(p_0) < S(q)$ and $S(q) < E(p_n)$, the incremental update (Dijkstra or Steele) write barrier on q can be safely eliminated.

3.3 Implementation-Specific Elimination

The fundamental reason for write barriers is to prevent loss of connectivity information when the mutator moves a pointer from an unscanned portion of the heap to a scanned portion of the heap. So far, we have assumed that any heap write may cause this to happen. However, if the collector is implemented in such a way as to guarantee certain traversal orders, additional barrier elimination may be possible.

For instance, assume that the collector always scans objects from low to high addresses. Then if a pointer were moved from a lower to a higher field or array element, a write barrier could be eliminated: for incremental update barriers, it would be the write barrier on the store to the new location, and for snapshot barriers, it would be the write barrier on the overwrite of the old location.

Such elimination conditions could also be applied if we certain traversal orders were statically provable, for instance of linked lists or trees.

However, applying such conditions may impose subtle additional complications. For instance, large arrays are often processed piecewise. In that case, it would be necessary to require that the pieces were processed in order. In addition, overflow of the marking stack may perturb the marking order.

4. RESULTS

In this section we present the results of our dynamic limit study. The results provide an upper bound on how many dynamic write barriers can be eliminated, by considering how many of them could be eliminated in a particular execution of a program.

We measured the potential for barrier removal based on both null pointers and on the elimination conditions for non-null pointers presented in the previous section. For elimination of non-null pointers, we measured the applicability of the single elimination conditions (SCC and SAC) only. When both conditions apply, we consider it as a case of SCC. This ensures that in our measurements any particular barrier can only be eliminated via one method: NULL, SCC, or SAC. In addition to providing an upper bound for each elimination condition, we study how various object attributes such as lifetime, popularity, and size relate to write barrier removal. We also measured how the potential for barrier elimination is correlated with time: whether it is bursty or continuous.

4.1 Traces and Methodology

We analyzed traces generated using an instrumented version of the Jikes RVM 2.2.0 Java virtual machine. The traces were gener-

Program	Write Barriers	Eliminated Barriers							
		Null	%	SCC	%	SAC	%	Total	%
Incremental Update (Dijkstra/Steele)									
power	23403	0	0.0	0	0.0	11835	50.6	11835	50.6
deltablue	3996808	133200	3.3	3077905	77.0	118155	3.0	3329260	83.3
bh	224676	40	0.0	68499	30.5	23244	10.4	91783	40.9
health	11671235	410547	3.5	10643082	91.2	54214	0.5	11107843	95.2
ipsixql	14214647	1806519	12.7	3274954	23.0	3004026	21.1	8085499	56.9
xalan	4423072	49881	1.1	162979	3.7	1636555	37.0	1849415	41.8
compress	1727	93	5.4	104	6.0	306	17.7	503	29.1
mpegaudio	143210	19	0.0	39434	27.5	1212	1.0	40665	28.4
db	33040494	674	0.0	26711999	80.9	83565	0.3	26796238	81.1
javac	13249090	1335482	10.1	3861753	29.2	1280108	9.7	6477343	48.9
mtrt	6122912	2287321	37.4	1115248	18.2	223346	3.7	3625915	59.2
jack	8848212	2043928	23.1	251517	2.8	212736	2.4	2508181	28.4
Snapshot (Yuasa)									
power	23403	23403	100.0	0	0.0	—	—	23403	100.0
deltablue	3996808	816506	20.4	2894180	72.4	—	—	3710686	92.8
bh	224676	135331	60.2	53968	24.0	—	—	189299	84.3
health	11671235	1639602	14.0	9948309	85.2	—	—	11587911	99.3
ipsixql	14214647	10667235	75.0	1341692	9.4	—	—	12008927	84.5
xalan	4423072	4308205	97.4	101265	2.3	—	—	4409470	99.7
compress	1727	1506	87.2	1	0.1	—	—	1507	87.3
mpegaudio	143210	1865	1.3	39353	27.5	—	—	41218	28.8
db	33040494	8496557	25.7	21548367	65.2	—	—	30044924	90.9
javac	13249090	9431706	71.2	2272659	17.2	—	—	11704365	88.3
mtrt	6122912	3761696	61.4	672986	11.0	—	—	4434682	72.4
jack	8848212	6147535	69.5	214138	2.4	—	—	6361673	71.9

Table 2: Eliminated Barriers

ated by Hirzel for use in [14]. We analyzed benchmarks from the SPECjvm98 suite [1] as well as other large-scale programs, summarized in Table 1. The traces contain the following heap events: thread creation, pointer assignments, object allocations, local/stack variable writes, static variable writes, and object death events. The death events are generated using the method of Hertz et al. [13].

There are three types of allocated objects in Jikes RVM: the Jikes RVM boot image objects, objects allocated by the running VM and objects allocated by the application. The boot image objects are allocated when the VM boots. Once the Jikes RVM has booted, the allocation site is used to classify objects into VM or application objects. If the allocation site is within the standard Java library, the dynamic chain is traversed until the caller is identified: either the RVM runtime system or the application. Since our primary goal is analyzing how applications behave, we have performed a dynamic analysis for application objects only.

Note that for the multi-threaded `mtrt` benchmark the potential for barrier elimination may be inflated, since some pointers may be covered by others purely as a result of the particular interleaving in the trace.

4.2 Elimination Results

Table 2 provides an upper bound on how many write barriers can be eliminated by utilizing the NULL condition and the SCC and SAC conditions. The table contains results for both incremental update (Dijkstra/Steele) and snapshot (Yuasa) write barriers. The first and second columns denote the application program together with the total number of application write barriers active. The third and fourth columns show how many barriers can be eliminated because

the pointer in question is NULL. The next four columns show the elimination potential of the single covering condition (SCC) and the single allocation condition (SAC). SAC is only applicable to incremental update barriers. The last two columns denote the total number of write barriers that can be eliminated by combining the NULL, SCC, and SAC conditions. Figures 7 and 8 present the information graphically for incremental update and snapshot barriers, respectively.

We observe that in all cases we can eliminate a greater number of snapshot barriers than incremental update barriers. By combining the results from the NULL and SCC conditions, there is the potential to eliminate over 71% of the snapshot barriers for all applications except `mpegaudio`. That is, with effective write barrier elimination, significantly fewer write barriers would be needed if a snapshot barrier were used. Snapshot-based collectors also have the advantage of much simpler and more efficient termination conditions.

On other hand, snapshot write barriers are more conservative and consequently suffer from more floating garbage. Snapshot barriers are also inherently more expensive because they must load the old pointer before over-writing it.

We also observe that a significant number of snapshot barriers can be eliminated by using only the NULL condition. But while this holds for the majority of benchmarks we measured, there are exceptions such as `deltablue`, `health` and `db`.

Snapshot barriers have another advantage in terms of barrier elimination because the NULL condition is far more prevalent, and we expect the NULL condition to be easier to prove than SCC or SAC. Since a significant number of snapshot barriers can be eliminated

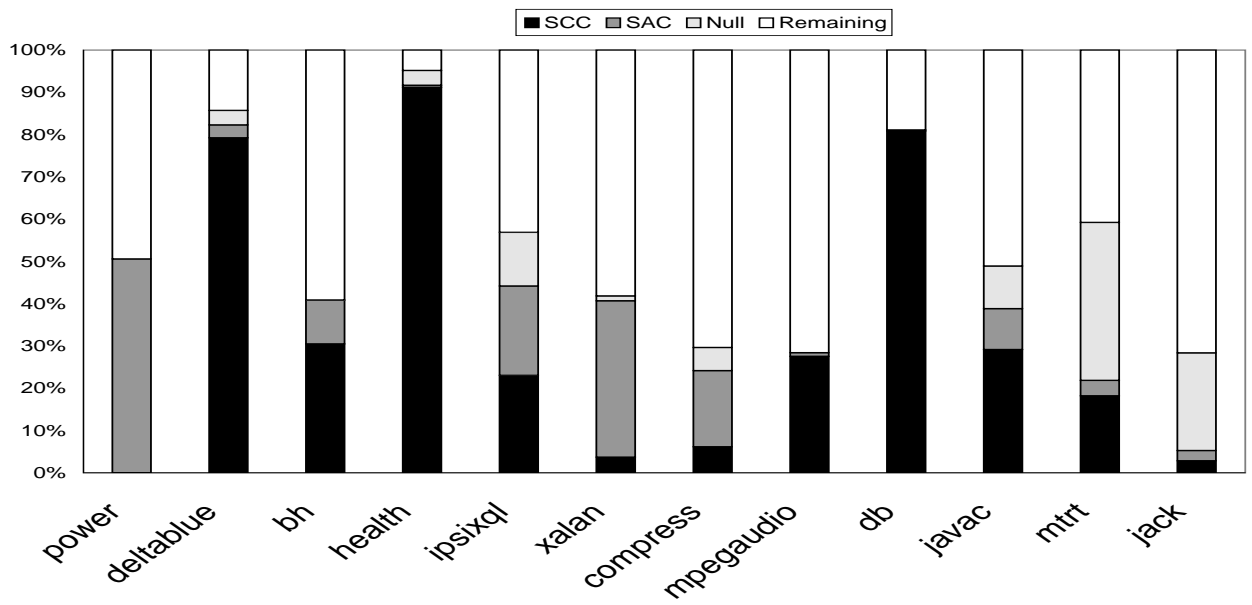


Figure 7: Eliminated Incremental Barriers

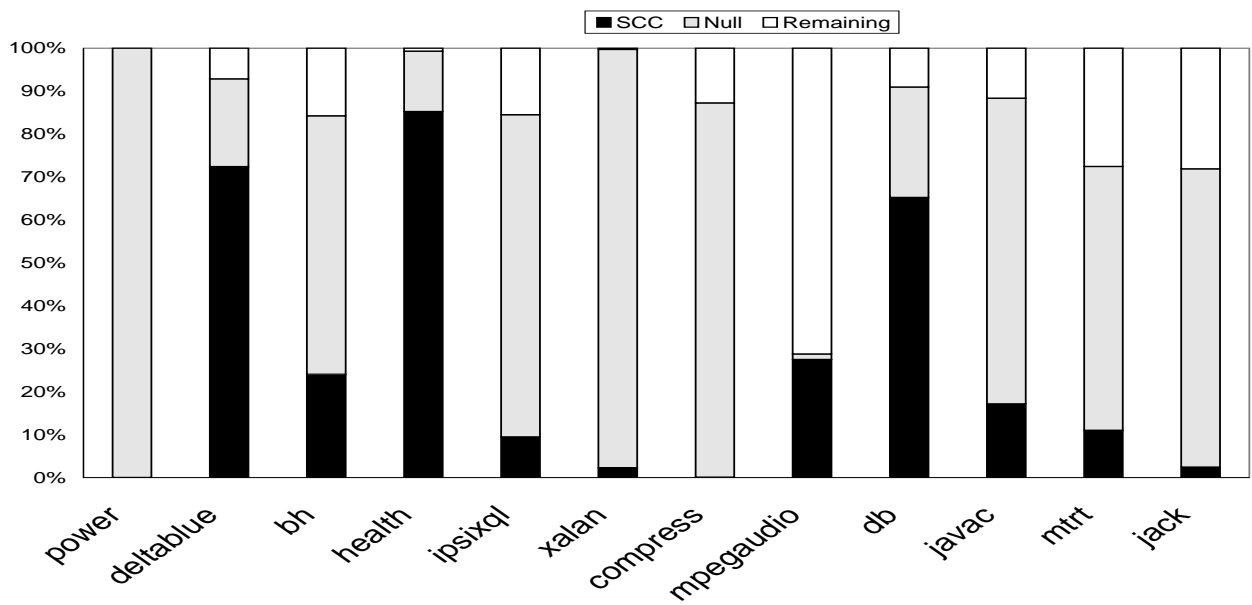


Figure 8: Eliminated Snapshot Barriers

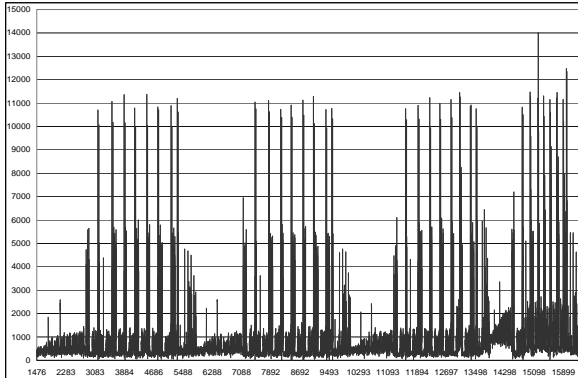


Figure 9: Time (in MB allocated) vs. Eliminated Snapshot Barriers (javac benchmark)

using the SCC and NULL conditions alone, a static analysis is unlikely to benefit from utilizing the MCC condition. That is, we cannot expect to eliminate significant number of barriers using the MCC. On the other side, in the case of incremental update barriers, a static analysis could possibly benefit by detecting the MAC and the MCC conditions. Therefore it is possible that the number of eliminated incremental barriers could increase significantly.

A subject for future study is to compare the overall effectiveness of incremental update-based collectors versus snapshot collectors, once write barrier elimination is performed. That is, to compare the total systemic effects of a snapshot collector’s increased floating garbage and more expensive write barriers, as opposed to an incremental update collector’s slower termination and increased number of write barriers.

4.3 Correlation

In this section we study various properties of write barrier elimination: how it correlates with object attributes such as size, lifetime and popularity and whether elimination occurs in bursts. We have selected two benchmarks *javac* and *db* which represent the spectrum of different results. The *javac* benchmark is representative of the majority of the benchmarks, while *db* represents program behavior which deviates from that of the other benchmarks. We present measurements for snapshot (Yuasa) barriers; the trends are the same for incremental update barriers.

4.3.1 Barrier Correlation Based on Time

It is important to understand whether barrier elimination happens in bursts or in continuous fashion. Such information could potentially guide the static analysis to consider only a small fraction of the methods executed. The dynamic compiler could effectively use this information to analyze only hot barrier methods, that is, methods where potentially a great number of barriers could be removed. Class file annotations could also be used effectively to hint the dynamic compiler of such methods based on profiling the program offline similarly to the method of Krintz and Calder [17].

Our results show that the potential for barrier elimination does indeed happen in bursts, and these results hold across all benchmarks. Both *javac* (Figure 9), which is typical of the applications and *db* (Figure 10), which is generally atypical, exhibit strong periodic behavior, suggesting the most write barriers can be elimi-

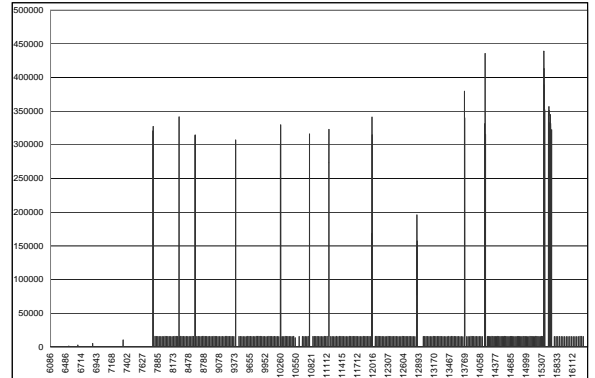


Figure 10: Time (in MB allocated) vs. Eliminated Snapshot Barriers (db benchmark)

nated by optimizing a small number of frequently executed loops or methods.

4.3.2 Barrier Correlation Based on Object Size

Figures 11 and 12 show that the vast majority of eliminated barriers protect small objects, which is consistent with the tendency of programs to allocate large numbers of small objects, and to mutate pointers to large objects slowly. This indicates that barrier elimination is not needed for objects residing in a segregated large object space.

It also suggests that barrier elimination can concentrate on a small number of prolific types. Such type-based approaches to limiting the work required of the optimizer are typically highly effective.

4.3.3 Barrier Correlation Based on Lifetime

Another approach is to concentrate on recently allocated objects. For instance, the generational write barrier elimination scheme of Zee and Rinard [29] only eliminates barriers on the most recently allocated object.

Most benchmarks that we measured showed a very strong correlation between barrier elimination and object youthfulness. Figure 13 is representative, showing that almost all eliminated barriers occur soon after allocation (note that the y-axis is logarithmic). On the other hand, Figure 14 shows results for the *db* benchmark, which is known not to obey the generational hypothesis, and indeed barrier elimination is also not concentrated on objects of short lifetime.

Another approach to barrier elimination is to partition the heap and omit barriers on a certain portion of the heap, using some other mechanism to ensure that the omitted portion is correctly scanned. For instance, if the concurrent collector is also generational, then we can omit barriers for objects in the nursery provided that the nursery is always scanned synchronously before collection terminates. The fact that barrier elimination is strongly correlated with youth suggests that such a generation-based elimination scheme could be highly effective.

4.3.4 Barrier Correlation Based on Popularity

Finally, we investigate how barrier removal correlates with object popularity. An object is *popular* if it is pointed to by many heap

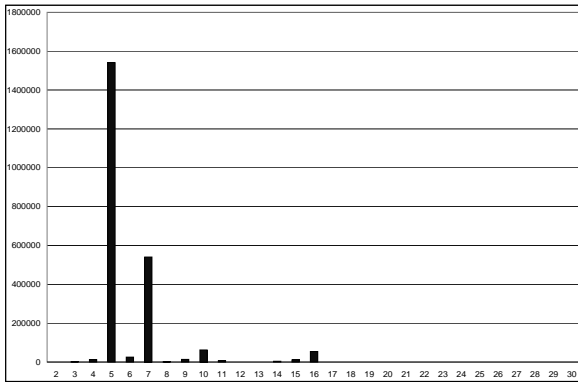


Figure 11: Object size (words) vs. Eliminated Snapshot Barriers (javac benchmark)

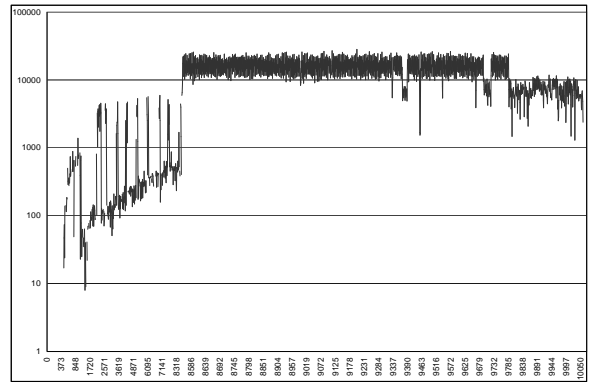


Figure 14: Object lifetime (in MB allocated) vs. Eliminated Snapshot Barriers (db benchmark). Y-axis is logarithmic.

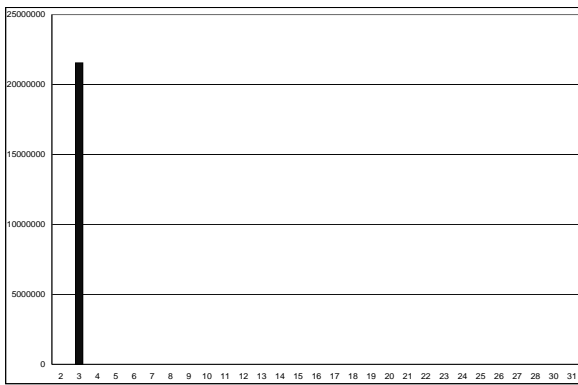


Figure 12: Object size (words) vs. Eliminated Snapshot Barriers (db benchmark)

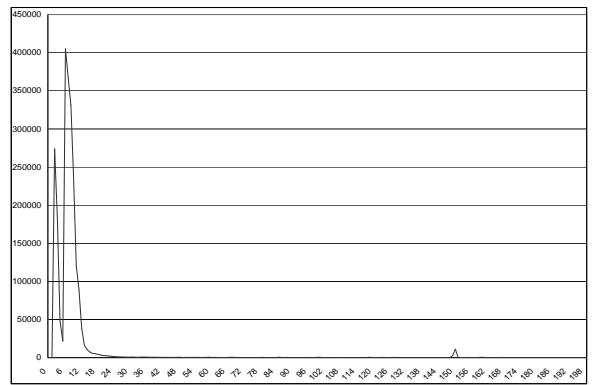


Figure 15: Incoming Pointers vs. Eliminated Snapshot Barriers (javac benchmark).

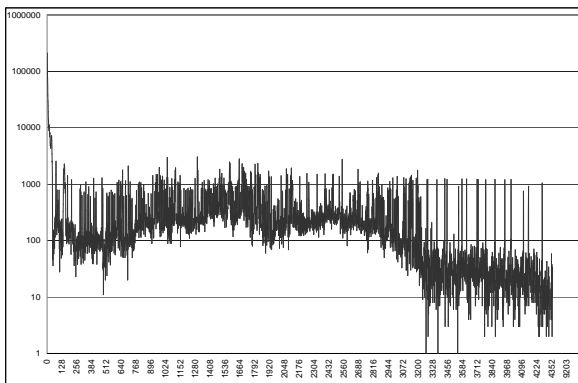


Figure 13: Object lifetime (in MB allocated) vs. Eliminated Snapshot Barriers (javac benchmark). Y-axis is logarithmic.

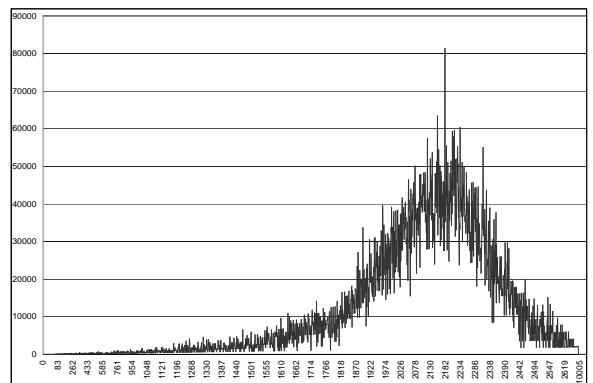


Figure 16: Incoming Pointers vs. Eliminated Snapshot Barriers (db benchmark).

pointers [16]. Hirzel et al. [15] have argued that there is no correlation between object lifetime and popularity. Our results indicate that in all of the benchmarks, a significant number of barriers are eliminated on great number of relatively unpopular objects. This result can be observed in Figure 15, which is representative of the overall results for the benchmarks used. Figure 16 reflects the exceptional behavior of the *db* benchmark.

Overall, our results indicate that in the majority of cases, write barrier elimination is concentrated on small objects of short lifetime with few incoming pointers.

Aside from the above mentioned properties, it would be interesting to evaluate how barrier elimination relates to recently introduced notions of prolific types and connectivity. We have not studied whether there is correlation between barrier elision and these properties.

5. RELATED WORK

Nandivada and Detlefs [19] have studied the compile-time elimination of redundant snapshot (Yuasa) barriers that always record null values (null elimination condition). They show that a significant number of such barriers can be eliminated by static analysis. This work is complementary to ours, which focuses primarily on elimination conditions for non-null pointers, and considers both incremental update as well as snapshot barriers.

Most prior work on barrier elimination has focused on eliminating write barriers for generational collectors. There has also been significant work studying the cost of barriers and ways to optimize them.

Zee and Rinard [29] observed that generational write barriers for the most recently allocated object are unnecessary, since it is known to reside in the nursery. Any pointers installed in a nursery object must point to either other nursery objects or to tenured objects. The paper presents several static analyses for eliding a write barrier based on this elimination condition. The approach is only suited for generational collectors.

Other than generational collectors, alternative heap partitioning schemes based on connectivity [15] and types [24, 23] have recently been proposed. A specific heap partitioning removes or partially eliminates the need for a generational write barrier between partitions. Nonetheless, the concurrent barrier would still have to be active if an on-the-fly collector was used.

Objects not allocated on the heap also do not require a write barrier. If objects are allocated in a region-like manner [26] using static or dynamic escape analysis techniques such as that of Qian and Hendren [22], the write barriers acting on such objects are unnecessary.

Zorn [30] was the first to systematically study the cost of both read and write barriers, for both generational and concurrent garbage collectors. Blackburn and McKinley [7] studied the cost on generational write barriers, with particular attention to the impact of various inlining levels. Blackburn and Hosking [6] provide a comprehensive study of both read and write barrier overheads in a modern dynamic Java compiler.

An approach to reducing the overhead of concurrent write barriers is to combine them with a generational barrier, when the collector is both concurrent and generational. This approach has been studied by Printezis and Detlefs [21]. Note that a write barrier would still have to be present at every heap write, but its combined overhead is less than if two separate barriers would have been used, that is one for generational and one for concurrent purposes. To remove such a combined barrier, an elimination condition must provide safety for both the generational and the concurrent components. The conditions in this paper deal with the concurrent aspect

while the condition presented in [29] could attack the generational portion of the write barrier.

In a concurrent collector that moves objects (usually for the purpose of compaction or defragmentation), a read barrier is required which usually has a direct cost that is significantly higher than that of the write barrier, since reads tend to outnumber writes by around five to one. Brooks [8] showed how the cost of such read barriers could be reduced by unconditionally following a forwarding pointer stored in the header of every object (at the cost of the extra space for the forwarding pointer). Bacon et al. [4] showed how the cost of a Brooks-style read barrier could be reduced by a combination of performing barrier operations “eagerly” and compiler optimizations that perform common subexpression elimination on read barriers.

6. CONCLUSIONS

We have identified two types of elimination conditions which can be utilized to elide write barriers for non-null pointers in concurrent garbage collectors. *Covering conditions* hold when a pointer or set of pointers has a lifetime that completely encompasses that of some other pointer. They can be applied to both incremental update (Dijkstra and Steele) barriers and to snapshot-at-the-beginning (Yuasa) barriers.

Allocation conditions can only be applied to incremental update barriers, and hold when the pointer to a newly allocated object is known to be on the stack when it is copied into the heap.

Each type of elimination condition (covering and allocation) has two variants: one in which the eliminated barrier is redundant because of the existence of a single pointer, and another in which the eliminated barrier is redundant because of the existence of a collection of pointers with overlapping lifetimes, that can together be considered as a “virtual pointer” from the standpoint of barrier elimination.

The elimination conditions are valid for all variations of Dijkstra, Steele and Yuasa introduced in the research literature so far.

We have presented the results of a limit study that evaluates the potential for the elimination of barriers either because the pointers are null or because our elimination conditions hold. The study only applied the single pointer variants of the elimination conditions.

Our results indicate that for both incremental update and snapshot barriers, a significant fraction can be elided based on our elimination conditions. The elimination conditions are slightly more effective for incremental update barriers. However, when combined with null pointer elimination, a much higher fraction of snapshot barriers can be eliminated, since many of these barriers are executed when the null pointers in newly allocated objects are overwritten. Using the combined elimination methods, it is possible to eliminate 71-100% of snapshot barriers for all but one of the benchmarks.

Of course, since this is a limit study, these statistics represent an upper bound. The actual achievable results will be lower due to both imprecision in static analysis and variations in dynamic behavior that are not captured in the traces.

Our measurements indicate that analysis overhead can be minimized by concentrating on barriers in frequently executed methods and for small, frequently allocated objects. This should help reduce the cost of the necessary static analysis.

The effectiveness of the barrier elimination conditions indicates that write barrier overhead, particularly for snapshot barriers, can be reduced by an order of magnitude. This will result in lower mutator overhead, faster garbage collection, less space consumption, and improved real-time properties.

Acknowledgements

Thanks are due to Martin Hirtzel for providing the dynamic traces upon we performed our measurements, to Peter Petrov, Tim Harris, Martin Richards and Alan Mycroft for the encouragement to proceed with the problem and for providing thoughtful feedback at various stages, to Umar Janjua and Alan Lawrence for the numerous discussions. We also thank David Grove, Perry Cheng, Mike Hind and the anonymous reviewers for their helpful comments which improved the paper.

7. REFERENCES

- [1] Specjvm98 benchmarks. In *Standard Performance Evaluation Corporation*.
- [2] ALPERN, B., ET AL. The Jalapeño virtual machine. *IBM Syst. J.* 39, 1 (Feb. 2000), 211–238.
- [3] AZATCHI, H., LEVANONI, Y., PAZ, H., AND PETRANK, E. An on-the-fly mark and sweep garbage collector based on sliding views. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Oct 2003), ACM Press, pp. 269–281.
- [4] BACON, D. F., CHENG, P., AND RAJAN, V. T. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of the 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New Orleans, Louisiana, Jan. 2003). *SIGPLAN Notices*, 38, 1, 285–298.
- [5] BARABASH, K., OSSIA, Y., AND PETRANK, E. Mostly concurrent garbage collection revisited. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Oct 2003), ACM Press, pp. 255–268.
- [6] BLACKBURN, S. M., AND HOSKING, A. L. Barriers: Friend or foe? In *Proc. of the International Symposium on Memory Management* (Vancouver, British Columbia, Oct. 2004).
- [7] BLACKBURN, S. M., AND MCKINLEY, K. S. In or out?: putting write barriers in their place. In *Proc. of the Third International Symposium on Memory management* (Berlin, Germany, June 2002). *SIGPLAN Notices*, 38, 2 (supplement) (Feb., 2003), 175–184.
- [8] BROOKS, R. A. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, Texas, Aug. 1984), G. L. Steele, Ed., pp. 256–262.
- [9] CHENG, P., AND BLELLOCH, G. E. A parallel, real-time garbage collector. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (Jun 2001), ACM Press, pp. 125–136.
- [10] DETLEFS, D., CLINGER, W., AND JACOB, M. Concurrent remembered set refinement in generational garbage collection. In *Usenix Java Virtual Machine Research and Technology Symposium* (2002).
- [11] DIJKSTRA, E., LAMPORT, L., MARTIN, A., SCHOLTEN, C., AND STEFFENS, E. On-the-fly garbage collection : an exercise in cooperation. In *Communications of the ACM* (1976).
- [12] DOMANI, T., KOLODNER, E. K., LEWIS, E., SALANT, E. E., BARABASH, K., LAHAN, I., LEVANONI, Y., PETRANK, E., AND YANORER, I. Implementing an on-the-fly garbage collector for java. In *Proceedings of the second international symposium on Memory management* (Oct 2000), ACM Press, pp. 155–166.
- [13] HERTZ, M., BLACKBURN, S. M., MOSS, J. E. B., MCKINLEY, K. S., AND STEFANOVIĆ, D. Error-free garbage collection traces: how to cheat and not get caught. In *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems* (Jun 2002), ACM Press, pp. 140–151.
- [14] HIRZEL, M., DIWAN, A., AND HERTZ, M. Connectivity-based garbage collection. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Oct 2003), ACM Press, pp. 359–373.
- [15] HIRZEL, M., HENKEL, J., DIWAN, A., AND HIND, M. Understanding the connectivity of heap objects. In *Proceedings of the third international symposium on Memory management* (Jun 2002), ACM Press, pp. 36–49.
- [16] HUDSON, R. L., AND MOSS, E. B. Incremental garbage collection for mature objects. In *Proc. of the International Workshop on Memory Management* (St. Malo, France, Sept. 1992), Y. Bekkers and J. Cohen, Eds., vol. 637 of *Lecture Notes in Computer Science*.
- [17] KRINTZ, C., AND CALDER, B. Using annotations to reduce dynamic optimization time. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation* (Jun 2001), ACM Press, pp. 156–167.
- [18] LEVANONI, Y., AND PETRANK, E. An on-the-fly reference counting garbage collector for java. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications* (Oct 2001), ACM Press, pp. 367–380.
- [19] NANDIVADA, V. K., AND DETLEFS, D. Compile-time concurrent marking write barrier removal. Submitted for publication, 2004.
- [20] PIRINEN, P. P. Barrier techniques for incremental tracing. In *Proceedings of the first international symposium on Memory management* (Oct 1998), ACM Press, pp. 20–25.
- [21] PRINTEZIS, T., AND DETLEFS, D. A generational mostly-concurrent garbage collector. In *Proceedings of the second international symposium on Memory management* (Oct 2000), ACM Press, pp. 143–154.
- [22] QIAN, F., AND HENDREN, L. An adaptive, region-based allocator for java. In *Proceedings of the third international symposium on Memory management* (Jun 2002), ACM Press, pp. 127–138.
- [23] SHUF, Y., GUPTA, M., BORDAWEKAR, R., AND SINGH, J. P. Exploiting prolific types for memory management and optimizations. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (Jan 2002), ACM Press, pp. 295–306.
- [24] SHUF, Y., GUPTA, M., FRANKE, H., APPEL, A., AND SINGH, J. P. Creating and preserving locality of java applications at allocation and garbage collection times. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Nov 2002), ACM Press, pp. 13–25.
- [25] STEELE, G. L. Multiprocessing compactifying garbage collection. *Commun. ACM* 18, 9 (Sept. 1975), 495–508.
- [26] TOFTE, M. A brief introduction to regions. In *Proceedings of the first international symposium on Memory management* (Oct 1998), ACM Press, pp. 186–195.
- [27] WILSON, P. Uniprocessor garbage collection techniques. In *Proceedings of the International Workshop on Memory Management* (1992).
- [28] YUASA, T. Real-time garbage collection on general purpose machines. In *Journal of Systems and Software* (1990).
- [29] ZEE, K., AND RINARD, M. Write barrier removal by static analysis. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Nov 2002), ACM Press, pp. 191–210.
- [30] ZORN, B. Barrier methods for garbage collection. Tech. Rep. CU-CS-494-90, University of Colorado at Boulder, 1990.