Eliminating Read Barriers through Procrastination and Cleanliness

KC Sivaramakrishnan Lukasz Ziarek Suresh Jagannathan
Purdue University
{chandras, lziarek, suresh}@cs.purdue.edu

Abstract

Managed languages typically use read barriers to interpret forwarding pointers introduced to keep track of copied objects. For example, in a multicore environment with thread-local heaps and a global, shared heap, an object initially allocated on a local heap may be copied to a shared heap if it becomes the source of a store operation whose target location resides on the shared heap. As part of the copy operation, a forwarding pointer may be established in the original object to point to the copied object. This level of indirection avoids the need to update all of the references to the object that has been copied.

In this paper, we consider the design of a managed runtime that eliminates read barriers. Our design is premised on the availability of a sufficient degree of concurrency to stall operations that would otherwise necessitate the copy. Stalled actions are deferred until the next local collection, avoiding exposing forwarding pointers to the mutator. In certain important cases, procrastination is unnecessary – lightweight runtime techniques can sometimes be used to allow objects to be eagerly copied when their set of incoming references is known, or when it can be determined that having multiple copies would not violate program semantics.

We evaluate our techniques on 3 platforms: a 16-core AMD64 machine, a 48-core Intel SCC, and an 864-core Azul Vega 3. Experimental results over a range of parallel benchmarks indicate that our approach leads to notable performance gains (20 - 32% on average) without incurring any additional complexity.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management – Allocation/Deallocation strategies, Garbage collection; D.3.3 [*Programming Languages*]: Language Constructs and Features – Concurrent programming structures

General Terms Algorithms, Design, Experimentation, Management, Measurement, Performance

Keywords barrier elimination, private heaps, parallel and concurrent collection, cleanliness, concurrent programming, functional languages

1. Introduction

Splitting a program heap among a set of cores is a useful technique to exploit available parallelism on scalable multicore platforms: each core can allocate, collect, and access data locally, moving objects to a global, shared heap only when they are accessed by threads executing on different cores. This design allows local heaps to be collected independently, with coordination required only for global heap collection. In contrast, stop-the-world collectors need a global synchronization for every collection. In order to ensure that cores cannot directly or indirectly access objects on other local heaps, which would complicate the ability to perform independent local heap collection, the following invariants need to be preserved:

- No pointers are allowed from one core's local heap to another.
- No pointers are permitted from the shared heap to the local heap.

Both invariants are necessary to perform independent local collections. The reason for the first is obvious. The second invariant prohibits a local heap from transitively accessing another local heap object via the shared heap. In order to preserve these invariants, the mutator typically executes a *write barrier* on every store operation. The write barrier ensures that before assigning a local object reference (source) to a shared heap object (target), the local object along with its transitive closure is lifted to the shared heap. We call such writes *exporting writes* as they export information out of local heaps. The execution of the write barrier creates *forwarding pointers* in the original location of the lifted objects in the local heap. These point to the new locations of the lifted objects in the shared heap. Since objects can be lifted to the shared heap on potentially any write, the mutator needs to execute a *read barrier* on potentially every read. The read barrier checks whether the object being read is the actual object or a forwarding pointer, and in the latter case, indirects to the object found on the shared heap. Forwarding pointers are eventually eliminated during local collection.

Because the number of reads are likely to far outweigh the number of writes, the aggregate cost of read barriers can be both substantial and vary dramatically based on underlying architecture characteristics [6]. Eliminating read barriers, however, is non-trivial. Abstractly, one can avoid read barriers by eagerly *fixing* all references that point to forwarded objects at the time the object is lifted to the shared heap, ensuring the mutator will never encounter a forwarded object. Unfortunately, this requires being able to enumerate all the references that point to the lifted object; in general, gathering this information is very expensive as the references to an object might originate from any object in the local heap.

In this paper, we consider an alternative design that completely eliminates the need for read barriers *without* requiring a full scan of the local heap whenever an object is lifted to the shared heap. The design is based on two observations. First, read barriers can be clearly eliminated if forwarding pointers are never introduced. One way to avoid introducing forwarding pointers is to *delay* operations that create them until a local garbage collection is triggered. In other words, rather than executing a store operation that would trigger lifting a thread local object to the shared heap, we can simply *procrastinate*, thereby stalling the thread that needs to perform the store. The garbage collector must simply be informed of the need to lift the object's closure during its next local collection. After collection is complete, the store can take place with the source object lifted, and all extant heap references properly adjusted. As long as there is sufficient concurrency to utilize existing computational resources, in the form of available runnable threads to run other computations, the cost of procrastination is just proportional to the cost of a context switch.

Second, it is not necessary to always stall an operation that involves lifting an object to the shared heap. We consider a new property for objects (and their transitive closures) called *cleanliness*. A clean object is one that can be safely lifted to the shared heap without introducing forwarding pointers that might be subsequently encountered by the mutator: objects that are immutable, whose elements are only referenced from the stack, or whose set of incoming heap references is known, are obvious examples. The runtime analysis for cleanliness is combined with a specialized write barrier to amortize its cost. Thus, procrastination provides a general technique to eliminate read barriers, while cleanliness serves as an important optimization that avoids stalling threads unnecessarily.

The effectiveness of our approach depends on a programming model in which (a) most objects are clean, (b) the transitive closure of the object being lifted rarely has pointers to it from other heap allocated objects, and (c) there is a sufficient degree of concurrency in the form of runnable threads; this avoids idling available cores whenever a thread is stalled performing an exporting write that involves an unclean object. In this paper, we consider an implementation of these ideas in the context of MultiMLton [17], a scalable, whole-program optimizing compiler and multicore-aware runtime system for Standard ML [15], a mostly functional language whose concurrent programs typically enjoy these properties. Our technique does not rely on programmer annotations, static analysis or compiler optimizations to eliminate read barriers, and can be completely implemented as a lightweight runtime technique.

This paper provides the following contributions:

- A garbage collector design that has been tuned for mostly functional languages in which there is typically a surfeit of concurrency (in the form of programmer-specified lightweight threads) available on each core, to realize a memory management system that does not require read barriers.
- A new object property called *cleanliness* that enables a certain (albeit broad) class of objects to be safely lifted to the shared heap without requiring a full traversal of the local heap to fix existing references to them, reducing the frequency of thread stalls as a result of procrastination.
- An extensive evaluation of the collector performance on three multicore platforms; a 16 core AMD Operton server, Intel's 48 core Single-chip Cloud Computer (SCC), and Azul System's 864 core Vega 3 processor. The results reveal that eliminating read barriers on these platforms can lead to significant performance improvements.

The paper is organized as follows. In the next section, we present additional motivation that quantifies the cost and benefit of read barriers in our system. The overall design and implementation of the collector is provided in Section 3. Section 4 describes our treatment of cleanliness. The modifications to our write barrier to support cleanliness analysis and delayed writes are presented in Section 5. Details about the target platforms we use in our experiments is given in Section 6. Experimental results are presented in Section 7. A comparison to related work is given in Section 8, and Section 9 presents conclusions.

```
pointer readBarrier (pointer p) {
  if (!isPointer(p)) return p;
  if (getHeader(p) == FORWARDED)
    return *(pointer*)p;
  return p;
}
```

Figure 1: Read barrier.

2. Motivation

In this section, we quantify the cost/benefit of read barriers in our system. The context of our investigation is a programming model that is mostly functional (our benchmarks are written in the asynchronous extension [24] of Concurrent ML [18]), and that naturally supports large numbers of lightweight user-level threads. We have implemented our garbage collector for MultiMLton [17], a parallel extension to MLton [16], that targets scalable, many-core platforms.

In our implementation, lightweight threads are multiplexed over kernel threads, with one kernel thread pinned to every core. Each core has a local heap, and a single shared heap is shared among all of the cores; the runtime system enforces the necessary heap invariants described earlier. In our experiments, we fixed the heap size to 3X the minimum heap size under which the programs would run.

MultiMLton performs a series of optimizations to minimize heap allocation, thus reducing the set of read barriers actually generated. For example, references and arrays that do not escape out of a function are flattened. Combined with aggressive inlining and simplification optimizations enabled by whole-program compilation, object allocation on the heap can be substantially reduced.

The compiler and runtime system ensure that entries on thread stacks never point to a forwarded object. Whenever an object pointer is stored into a register or the stack, a read barrier is executed on the object pointer to get the current location of the object. Immediately after an exporting write or a context switch, the current stack is walked and references to forwarded objects are updated to point to the new location of lifted objects in the shared heap. Additionally, before performing an exporting write, register values are saved on the stack, and reloaded after exit. Thus, as a part of fixing references to forwarding pointers from the stack, references from registers are also fixed. This ensures that the registers never point to forwarded objects either. Hence, no read barriers are required for dereferencing object pointers from the stack or registers. This optimization is analogous to "eager" read barriers as described in [4]. Eager read barrier elimination has marked performance benefits for repeated object accesses, such as array element traversals in a loop, where the read barrier is executed once when the array location is loaded into a register, but all further accesses can elide executing the barrier.

Whenever an object is lifted to the shared heap, the original object's header is set to `FORWARDED` and the first word of the object is overwritten with the new location of the object in the shared heap. Before an object is read, the mutator checks whether the object has been forwarded, and if it is, returns the new location of the object. Hence, our read barriers are conditional [5, 6].

Figure 1 shows the pseudo-C code for our read barrier. MLton represents non-value carrying constructors of (sum) datatypes us-

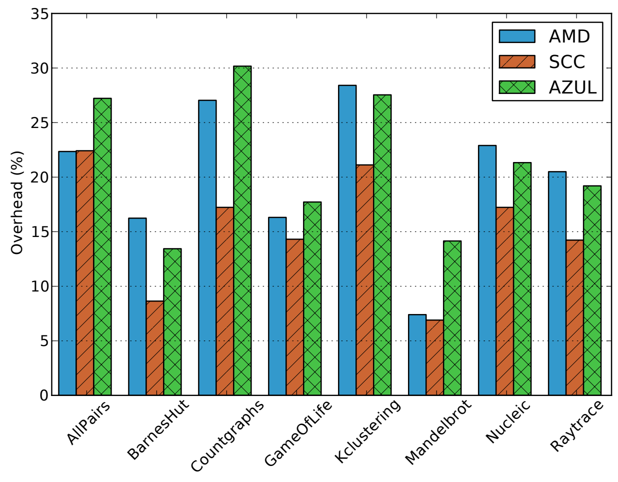


Figure 2: Read barrier overhead as a percentage of mutator time.

Benchmark	AllPairs	BarnesHut	Countgraphs	GameOfLife	Kclustering	Mandelbrot	Nucleic	Raytrace
Checks (X 10^6)	9,753	2,864	2,584	4,858	3,780	2,980	2,887	2,217
Forwarded	123	52702	0	2143	101	23	328	0

Figure 3: Effectiveness of read barrier checks: Checks represents the number of read barrier invocations and forwarded represents the number of instances when the read barrier encountered a forwarded object.

ing non-pointer values. If such a type additionally happens to have value-carrying constructors that reference heap-allocated objects, the non-pointer value representing the empty constructor will be stored in the object pointer field. Hence, the read barrier must first check whether the presumed pointer does in fact point to a heap object. Otherwise, the original value is returned (line 2). If the given pointer points to a forwarded object, the current location of the object stored is returned. Otherwise, the original value is returned.

We evaluated a set of 8 benchmarks (described in Section 7.1) running on a 16 core AMD64, a 48 core Intel SCC and an 864 core Azul Vega 3 machine to measure read barrier overheads. Figure 2 shows these overheads as a percentage of mutator time. Our experiments reveal that, on average, the mutator spends 20.1%, 15.3% and 21.3% of time executing read barriers on the AMD64, SCC and Azul architectures, respectively, for our benchmarks.

Although a Brooks-style unconditional read barrier would have avoided the cost of the second branch in our read barrier implementation, it would necessitate having an additional address length field in the object header for an indirection pointer. Most objects in our system tend to be small. In our benchmarks, we observed that 95% of the objects allocated were less than 3 words in size, including a word-sized header. The addition of an extra word in the object header for an indirection pointer would lead to substantial memory overheads, which in turn leads to additional garbage collection costs. Hence, we choose to encode read barriers conditionally rather than unconditionally.

But, does the utility of the read barrier justify its cost? We measure the number of instances the read barrier is invoked and the number of instances the barrier finds a forwarded object (see Figure 3). We see that read barriers find forwarded objects in less than one thousands of a percent of the number of instances they are invoked. Thus, in our system, the cost of read barriers is substantial, but only rarely do they have to perform the task of forwarding references. These results motivate our interest in a memory management design that eliminates read barriers altogether.

3. GC Design and Implementation

In this section, we describe the design and implementation of the runtime system and garbage collector.

3.1 Threading system

Our programming model separates program-level concurrency from the physical parallelism available in the underlying machine through the use of lightweight, user-level threads. These lightweight threads are multiplexed over system-level threads. One system-level thread is created for every core and is pinned to it. Thus, the runtime system effectively treats a system-level thread as a virtual processor. Load distribution is through work sharing, where threads are eagerly spawned on different cores in a round-robin fashion. Once created on a core, lightweight threads never migrate to another core.

Lightweight threads are preemptively scheduled on every core. On a timer interrupt, the threading system is informed that an interrupt has occurred by setting a flag at a known location. At every garbage collector safe-point, the current thread checks whether the timer interrupt flag has been set, and if it is, resets the flag and yields control to another thread.

3.2 Baseline collector (Stop-the-world)

The baseline heap design uses a single, contiguous heap, shared among all cores. In order to allow local allocation, each core requests a page-sized chunk from the heap. While a single lock protects the chunk allocation, objects are allocated within chunks by bumping a core-local heap frontier.

In order to perform garbage collection, all the cores synchronize on a barrier, with one core responsible for collecting the entire heap. The garbage collection algorithm is inspired from Sansom's [19] collector, which combines Cheney's two-space copying collector and Jonker's single-space sliding compaction collector. Cheney's copying collector walks the live objects in the heap just once per collection, while Jonker's mark-compact collector performs two walks. But Cheney's collector can only utilize half of memory allocated for the heap. Sansom's collector combines the best of both worlds. Copying collection is performed when heap requirements are less than half of the available memory. The runtime system dynamically switches to mark-compact collection if the heap utilization increases beyond half of the available space.

Since ML programs tend to have a high rate of allocation, and most objects are short-lived temporaries, it is beneficial to perform generational collection. The garbage collector supports Appel-style generational collection [2] for collecting temporaries. The generational collector has two generations, and all objects that survive a generational collection are copied to the older generation. Generational collection can work with both copying and mark-compact major collection schemes. The runtime system chooses to perform generational collection if the ratio of live objects to the total objects falls below a tunable threshold.

Our choice of a stop-the-world baseline collector was to enable better understanding of mutator overheads among various local collector designs, as opposed to illustrating absolute performance improvement of the local collectors over the baseline. Although a parallel collector would have improved overall baseline performance, we would expect poorer scalability due to frequent global synchronizations [10, 14, 20].

3.3 Local collector (Split-heap)

As mentioned earlier, the local collector operates over a single shared (global) heap and a local heap for each core. The allocation of the shared heap is performed similar to allocations in the stop-the-world collector, where each core allocates a page-sized chunk in the shared heap and performs object allocation by bumping its core-local shared heap frontier. Allocations in the local heaps do not require any synchronization. Garbage collection in the local heaps is similar to the baseline collector, except that it does not require global synchronization.

Objects are allocated in the shared heap only if they are to be shared between two or more cores. Objects are allocated in the shared heap because of exporting writes and remote spawns (Section 5.3). Apart from these, all globals are allocated in the shared heap, since globals are visible to all cores by definition. For a shared heap collection, all of the cores synchronize on a barrier and then a single core collects the heap. Moreover, along with globals, all the references from local heaps are considered to be roots for a shared heap collection. In order to eliminate roots from dead local heap objects, before a shared heap collection, local collections are performed on each core to eliminate such references.

The shared heap is also collected using Sansom's dual-mode garbage collector. However, we do not perform generational collection on the shared heap. This is because shared heap collection is expected to be relatively infrequent when compared to the frequency of local heap collections, and objects that are shared between cores, in general, live longer than a typical object collected during a generational collection.

3.3.1 Remembered stacks

In our system, threads can synchronously communicate with each other over first-class message-passing communication channels. If a receiver is not available, a sender thread can block on a channel. If the channel resides in the shared heap, the thread object, its associated stack and the transitive closure of all objects reachable from it on the heap would be lifted to the shared heap as part of the blocking action. Since channel communication is the primary mode of thread interaction in our system, we would quickly find that most local heap objects end up being lifted to the shared heap. This would be highly undesirable.

Hence, we choose never to move stacks to the shared heap. We add an exception to our heap invariants to allow thread $\rightarrow$ stack pointers, where the thread resides on the shared heap, and references a stack object found on the local heap. Whenever a thread object is lifted to the shared heap, a reference to the corresponding stack object is added to the set of remembered stacks. This remembered set is considered as a root for a local collection to enable tracing of remembered stacks.

Before a shared heap collection, the remembered set is cleared; only those stacks that are reachable from other GC roots survive the shared heap collection. After a shared heap collection, the remembered set of each core is recalculated such that it contains only those stacks, whose corresponding thread objects reside in the shared heap, and have survived the shared heap collection.

4. Cleanliness Analysis

In this section, we describe our cleanliness analysis. We first present auxiliary definitions that will be utilized by cleanliness checks.

4.1 Heap session

Objects are allocated in the local heap by bumping the local heap frontier. In addition, associated with each local heap is a pointer called `sessionStart` that always points to a location between the start of the heap and the frontier. We introduce the idea of a *heap session*, to capture the notion of recently allocated objects. Every local heap has exactly two sessions: a *current session* between the `sessionStart` and the heap frontier and a *previous session* between the start of the heap and `sessionStart`. Heap sessions are used by the cleanliness analysis to limit the range of heap locations that need to be scanned to test an object closure[1] for cleanliness. A new session can be started by setting the `sessionStart` to the current local heap frontier. We start a new session on a context switch, a local garbage collection and after an object has been lifted to the shared heap.

```
 1 Val writeBarrier (Ref r, Val v) {
 2   if (isObjptr(v)) {
 3     //Lift if clean or procrastinate
 4     if (isInSharedHeap(r) &&
 5         isInLocalHeap(v)) {
 6       needsFixup = false;
 7       if (isClean(v, &needsFixup))
 8         v = lift(v, needsFixup);
 9       else
10         v = suspendTillGCAndLift(v);
11     }
12     //Tracking cleanliness
13     if (isInLocalHeap (r) &&
14         isInLocalHeap(v)) {
15       n = getRefCount(v);
16       if (!isInCurrentSession (r))
17         setNumRefs(v, GLOBAL);
18       else if (n == ZERO)
19         setNumRefs(v, ONE);
20       else if (n < GLOBAL)
21         setNumRefs(v, LOCAL_MANY);
22     }
23   }
24   return v;
25 }
```

Figure 4: Write barrier implementation.

4.2 Reference count

We introduce a limited reference counting mechanism for local heap objects that counts the number of references from other local heap objects. Importantly, we do not consider references from ML thread stacks. The reference count is meaningful only for objects reachable in the current session. For such objects, the number of references to an object can be one of four values: `ZERO`, `ONE`, `LOCAL_MANY`, and `GLOBAL`. We steal 2 bits from the object header to record this information. A reference count of `ZERO` indicates that the object only has references from registers or stacks, while an object with a count of `ONE` has exactly one pointer from the current session. A count of `LOCAL_MANY` indicates that this object has more than one reference, but that all of these references originate from the current session. `GLOBAL` indicates that the object has at least one reference that originates from outside the current session.

The reference counting mechanism is implemented as a part of the write barrier. Lines 13–22 in Figure 4 illustrate the implementation of the reference counting mechanism, and Figure 5 illustrates the state transition diagram for the reference counting mechanism. Observe that reference counts are non-decreasing. Hence, the ref-

[1] In the following, we write *closure* (in the absence of any qualification) to mean the set of objects reachable from some root on the heap; to avoid confusion, we write *function closure* to mean the representation of an SML function as a pair of function code pointer and static environment.

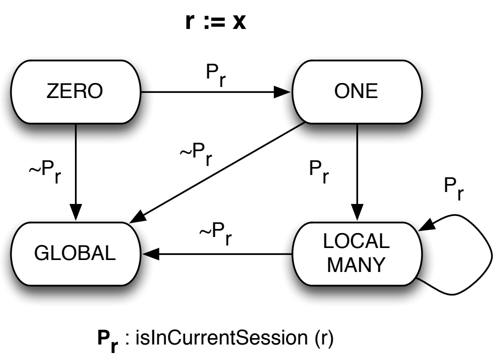


Figure 5: State transition diagram detailing the behavior of the reference counting mechanism with respect to object `x` involved in an assignment, `r := x`, where P_r = `isInCurrentSession(r)`.

```
1  bool isClean (Val v, bool* needsFixup) {
2    clean = true;
3    foreach o in reachable(v) {
4      if (!isMutable(o) || isInSharedHeap(o))
5        continue;
6      nv = getRefCount(o);
7      if (nv == ZERO)
8        clean &= true;
9      else if (nv == ONE)
10       clean &= (o != v);
11     else if (nv == LOCAL_MANY) {
12       clean &= (isInCurrentSession(o));
13       *needsFixup = true;
14     }
15     else
16       clean = false;
17   }
18   return clean;
19 }
```

Figure 6: Cleanliness check.

erence count of any object represents the maximum number of references that pointed to the object at any point in its lifetime.

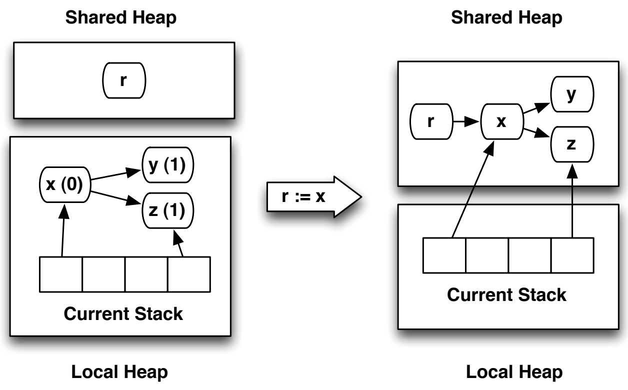
4.3 Cleanliness

An object closure is said to be clean, if for each object reachable from the root of the closure,

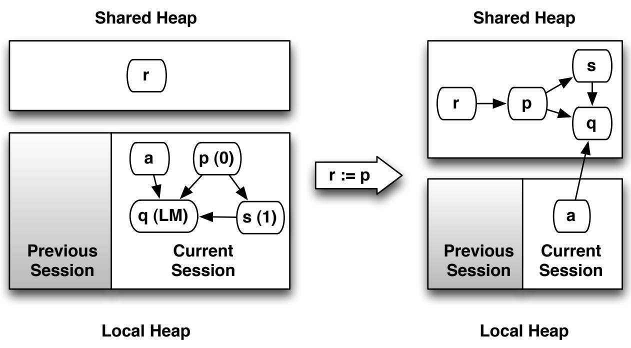
- the object is immutable or in the shared heap. Or,
- the object is the root, and has `ZERO` references. Or,
- the object is not the root, and has `ONE` reference. Or,
- the object is not the root, has `LOCAL_MANY` references, and is in the current session.

Otherwise, the object closure is not clean. Figure 6 shows an implementation of an object closure cleanliness check.

If the source of an exporting assignment is immutable, we can make a copy of the immutable object in the shared heap, and avoid introducing references to forwarded objects. Unlike languages like Java or C#, Standard ML does not allow the programmer to test the referential equality of immutable objects. Equality of immutable objects is always computed by structure. Hence, it is safe to replicate immutable objects. If the object is already in the shared heap, there is no need to move this object.



(a) Tree-structured object closure



(b) Session-based cleanliness

Figure 7: Utilizing closure cleanliness information for exporting writes to avoid references to forwarded objects.

If the object closure of the source of a exporting write is clean, we can move the closure to the shared heap and quickly fix all of the forwarding pointers that might be generated. For example, consider an object that defines a tree structure; such an object is clean if the root has `ZERO` references and all of its internal nodes have `ONE` reference from their parent. A root having `ZERO` references means it is accessed only via the stack; if it had a count of `ONE`, the outstanding reference may emanate from the heap. Internal nodes having a reference count of `ONE` implies they are reachable only via other nodes in the object being traced. Figure 7a shows such a closure. In this example, we assume that all objects in the closure are mutable. The reference count of relevant nodes is given in the brackets. Both the root and internal nodes can have pointers from the current stack not tracked by the reference count. After lifting the closure, the references originating from the current stack are fixed by walking the stack.

But, object closures need not just be trees and can be arbitrary graphs, with multiple incoming edges to a particular object in the closure. How do we determine if the incoming edges to an object originate from the closure or from outside the closure (from the local heap)? We cannot answer this question without walking the local heap. Hence, we simplify the question to asking whether all the pointers to an object originate from the current session. This question is answered in the affirmative if an object has a reference count of `LOCAL_MANY` (lines 11–13 in Figure 6).

Figure 7b shows an example of a closure whose objects have at most `LOCAL_MANY` references. Again, we assume that all objects in the closure are mutable. In the transitive closure rooted at `p`, object `q` has locally many references. These references might originate from the closure itself (edges `p` $\longrightarrow$ `q` and `s` $\longrightarrow$ `q`) or from outside the closure (edge `a` $\longrightarrow$ `q`). After lifting such closures to the shared heap, only the current session is walked to fix all of the references to forwarded objects created during the copy. In practice (Section 7.5), current session sizes are much smaller than heap sizes, and hence exporting writes can be performed quickly.

5. Write barrier

In this section, we present the modifications to the write barrier to eliminate the possibility of creating references from reachable objects in the local heap to a forwarded object. The implementation of our write barrier is presented in Figure 4. A write barrier is invoked prior to a write and returns a new value for the source of the write. The check `isObjptr` at line 2 returns true only for heap allocated objects, and is a compile time check. Hence, for primitive valued writes, there is no write barrier. Lines 4 and 5 check whether the write is exporting. If the source of the object is clean, we lift the transitive object closure to the shared heap and return the new location of the object in the shared heap.

5.1 Delaying writes

If the source of an exporting write is not clean, we suspend the current thread and switch to another thread in our scheduler. The source of the write is added to a queue of objects that are waiting to be lifted. Since the write is not performed, no forwarded pointers are created. If programs have ample amounts of concurrency, there will be other threads that are waiting to be run. However, if all threads on a given core are blocked on a write, we move all of the object closures that are waiting to be lifted to the shared heap. We then force a local garbage collection, which will, as a part of the collection, fix all of the references to point to the new (lifted) location on the shared heap. Thus, the mutator never encounters a reference to a forwarded object.

5.2 Lifting objects to the shared heap

Figure 8 shows the pseudo-C code for lifting object closures to the shared heap. The function `lift` takes as input the root of a clean object closure and a Boolean representing whether the closure has any object that has `LOCAL_MANY` references. For simplicity of presentation, we assume that the shared heap has enough space reserved for the transitive closure of the object being lifted. In practice, the lifting process requests additional shared heap chunks to be reserved for the current processor, or triggers a shared heap collection if there is no additional space in the shared heap.

Objects are transitively lifted to the shared heap, starting from the root, in the obvious way (Lines 22–24). As a part of lifting, mutable objects are lifted and a forwarding pointer is created in their original location, while immutable objects are copied and their location added to `imSet` (Lines 10–15). After lifting the transitive closure of the object to the shared heap, the shared heap frontier is updated to the new location.

After object lifting, the current stack is walked to fix any references to forwarding pointers (Line 27–28). Since we do not track references from the stack for reference counting, there might be references to forwarded objects from stacks other than the current stack. We fix such references lazily. Before a context switch, the target stack is walked to fix any references to forwarded objects. Since immutable objects are copied and mutable objects lifted, a copied immutable object might point to a forwarded object. We walk all the shared heap copies of immutable objects lifted from the local

```
1  Set imSet;
2  void liftHelper (pointer* op,
3                   pointer* frontierP) {
4    frontier = *frontierP;
5    o = *op;
6    if (isInSharedHeap(o)) return;
7    copyObject (o, frontier);
8    *op = frontier + headerSize(o);
9    *frontierP = frontier + objectSize(o);
10   if (isMutable(o)) {
11     setHeader(o, FORWARDED);
12     *o = *op;
13   }
14   else
15     imSet += o;
16 }
17
18 pointer lift (pointer op, bool needsFixup) {
19   start = frontier = getSharedHeapFrontier();
20   imSet = {};
21   //Lift transitive closure
22   liftHelper (&op, &frontier);
23   foreachObjptrInRange
24     (start, &frontier, liftHelper);
25   setSharedHeapFrontier(frontier);
26   //Fix forwarding pointers
27   foreachObjptrInObject
28     (getCurrentStack(), fixFwdPtr);
29   foreach o in imSet
30     foreachObjptrInObject(o, fixFwdPtr);
31   frontier = getLocalHeapFrontier();
32   if (needsFixup)
33     foreachObjptrInRange(getSessionStart(),
34                          &frontier, fixFwdPtr);
35   setSessionStart(frontier);
36   return op;
37 }
```

Figure 8: Lifting an object closure to the shared heap.

```
ThreadID spawn (pointer closure, int target) {
  ThreadID tid = newThreadID();
  Thread t = newThread(closure, tid);
  needsFixup = false;
  if (isClean(t, &needsFixup)) {
    t = lift(t, needsFixup);
    enqueThread(t, target);
  }
  else
    liftAndReadyBeforeGC(t, target);
  return tid;
}
```

Figure 9: Spawning a thread.

heap to fix any references to forwarded objects (Lines 29–30). If there were `LOCAL_MANY` references to any object in the lifted closure, the local session is walked to fix the references to forwarding pointers. Finally, session start is moved to the current frontier.

5.3 Remote spawns

Apart from exporting writes, function closures can also escape local heaps when threads are spawned on other cores. For spawning on other cores, the environment of the function closure is lifted to the shared heap and then, the function closure is added to the target core's scheduler. This might introduce references to forwarding pointers in the spawning core's heap. We utilize the techniques developed for exporting writes to handle remote spawns in a similar fashion.

Figure 9 shows the new implementation of thread spawn. If the function closure is clean, we lift the function closure to the shared heap, and enqueue the thread on the target scheduler. Otherwise, we add it to the list of threads that need to be lifted to the shared heap. Before the next garbage collection, these function closures are lifted to the shared heap, enqueued to target schedulers, and the references to forwarded objects are fixed as a part of the collection. When the target scheduler finds this new thread (as opposed to other preempted threads), it allocates a new stack in the local heap. Hence, except for the environment of the remotely spawned thread, all data allocated by the thread is placed in the local heap.

5.4 Barrier implementation

For our evaluation, we have implemented two local collector designs; one with read barriers (RB+ GC) and the other without read barriers incorporating the proposed techniques (RB- GC). Read barriers are generated as part of RSSA, one of the backend intermediate passes in our compiler. RSSA is similar to Static Single Assignment (SSA), but exposes data representations decisions. In RSSA, we are able to distinguish heap allocated objects from non-heap values such as constants, values on the stack and registers, globals, etc. This allows us to generate barriers only when necessary.

Although the code for tracking cleanliness is implemented as an RSSA pass (Lines 13–24 in Figure 4), the code for avoiding creation of references to forwarded objects (Lines 4–11 in Figure 4) is implemented in the primitive library, which has access to the lightweight thread scheduler. `suspendTillGCAndLift` (line 11 in Figure 4) is carefully implemented to not contain an exporting write, which would cause non-terminating recursive calls to the write barrier.

6. Target Architectures

We have implemented our GC design on three different architectures; a 16-core AMD64 running Linux (AMD), a 48-core Intel Single-chip Cloud Computer (SCC), and an 864-core Azul's Vega 3 machine. Our choice of architectures is primarily to study the robustness of our techniques across various architectures rather than exploiting the fine-grained architectural characteristics for our design.

The AMD machine has 8 dual core AMD Opteron processors, with each core running at 1.8 GHz. Each core has 64 KB of 2-way associative L1 data and instruction caches, and 1 MB of exclusive 16-way associative L2 cache with 32 GB of main memory. The peak memory bandwidth for serial access is 1.5 GB/s and 680 MB/s for all cores accessing the memory in parallel. These memory bandwidth numbers were measured using the STREAM [23] benchmark.

The Azul machine used in our experiments has 16 Vega 3 processors, each with 54-cores per chip; each core exhibits roughly 1/3 the performance of an Intel Core2-Duo. Out of the 864 cores, 846 are application usable while the rest of the cores are reserved for the kernel. The machine has 384 GB of cache coherent memory split across 192 memory modules. Uniform memory access is provided through a passive, non-blocking interconnect mesh. The machine has 205 GB/s aggregate memory bandwidth and 544 GB/s aggregate interconnect bandwidth. Each core has a 16KB, 4-way L1 data and instruction caches.

Intel's Single-chip Cloud Computer (SCC)[12] is an experimental platform from Intel labs with 48 P54C Pentium cores. The most interesting aspect of SCC is the complete lack of cache coherence and a focus on inter-core interactions through a high speed mesh interconnect. The cores are grouped into 24 tiles, connected via a fast on-die mesh network. The tiles are split into 4 quadrants with each quadrant connected to a memory module. Each core has 16KB L1 data and instruction caches and 256KB L2 cache. Each core also has a small message passing buffer (MPB) of 8KB used for message passing between the cores.

Since the SCC does not provide cache coherence, coherence must be implemented in software if required. From the programmer's perspective, each core has a private memory that is cached and not visible to other cores. The cores also have access to a shared memory, which is by default not cached to avoid coherence issues. The cost of accessing data from the cached local memory is substantially less when compared to accessing shared memory. It takes 18 core cycles to read from the L2 cache; on the other hand, it takes 40 core cycles to request data from the memory controller, 4 mesh cycles for the mesh to forward the request and 46 memory cycles for the memory controller to complete the operation. Hence, in total, the delay between a core requesting data from the memory controller is $40\ k_{core}\ +\ 4 * 2 * n\ k_{mesh}\ +\ 46\ k_{ram}$ cycles, where $k_{core},\ k_{mesh}\ and\ k_{ram}$ are the cycles of core, mesh network and memory respectively. In our experimental setup, where 6 tiles share a memory controller, the number of hops n to the memory controller could be $0 < n < 5$. Hence, shared heap accesses are much more expensive than local heap accesses.

6.1 Local collector on SCC

We briefly describe our runtime system design for the SCC. On the SCC, each core runs a Linux operating system and from the programmer's point-of-view, SCC is exposed as a cluster of machines. Thus, we believe that our local collector design is a must for circumventing coherence and segmentation restrictions, and making effective use of the memory hierarchy. Pointers to local memory are sensible only to the owning core. From the perspective of other cores, pointers might fall outside the segmentation boundary. If we were to utilize a single-shared heap design, where any object can point to any other object in the heap, the heap would have to be placed in the non-cached shared memory because of the lack of coherence.

Instead of spawning threads to represent virtual processors, we spawn one process on each core. Local heaps are placed in the cached private memory while the shared heap is placed in the non-cached shared memory. Since our local collector design only exports objects to the shared heap if they are to be shared between cores, most access are from the local heap and are cached. We modify the memory manager such that the shared heap is created at the same virtual address on each core. This avoids address translation overheads (and hence, read barriers) for shared heap reads.

Shared heap collection is *collective*; the collection proceeds in SPMD mode with each processor collecting roots from its local heap, followed by a single core collecting the shared heap. Finally, each core updates the references from its local heap to the shared heap with the new location of the shared heap object. The MPB is utilized by shared heap collection for synchronization and data exchange.

7. Results

7.1 Benchmarks

The benchmarks shown in Figure 10 were designed such that the input size and the number of threads are tunable; each of these benchmarks were derived from a sequential standard ML implementation, and parallelized using our lightweight thread system and CML-style [18] message-passing communication.

Benchmark	Allocation Rate (MB/s)			Bytes Allocated (GB)				# Threads		
	AMD	SCC	AZUL	AMD	SCC	AZUL	% Sh	AMD	SCC	AZUL
AllPairs	817	53	1505	16	16	54	11	256	512	32768
Barneshut	772	70	1382	20	20	876	2	512	1024	32768
Countgraphs	2594	144	4475	24	24	1176	1	128	256	16384
GameOfLife	2445	127	4266	21	21	953	13	256	1024	8192
Kclustering	3643	108	8927	32	32	1265	3	256	1024	8192
Mandelbrot	349	43	669	2	2	32	8	128	512	8192
Nucleic	1430	87	4761	13	14	609	1	64	384	16384
Raytrace	809	54	2133	11	12	663	4	128	256	2048

Figure 10: Benchmark characteristics. %Sh represents the average fraction of bytes allocated in the shared heap across all the architectures.

- **AllPairs**: an implementation of Floyd-Warshall algorithm for computing all pairs shortest path.
- **BarnesHut**: an n-body simulation using Barnes-Hut algorithm.
- **CountGraphs**: computes all symmetries (automorphisms) within a set of graphs.
- **GameOfLife**: Conway's Game of Life simulator
- **Kclustering**: a k-means clustering algorithm, where each stage is spawned as a server.
- **Mandelbrot**: a Mandelbrot set generator.
- **Nucleic**: Pseudoknot [11] benchmark applied on multiple inputs.
- **Raytrace**: a ray-tracing algorithm to render a scene.

Parameters are appropriately scaled for different architectures to ensure sufficient work for each of the cores. The benchmarks running on AMD and SCC were given the same input size. Hence, we see that the benchmarks allocate the same amount of memory during their lifetime. But, we increase the number of threads on the SCC when compared to AMD since there is more hardware parallelism available. For Azul, we scale both the input size and the number of threads, and as a result we see a large increase in bytes allocated when compared to the other platforms. Out of the total bytes allocated during the program execution, on average 5.4% is allocated in the shared heap. Thus, most of the objects allocated are collected locally, without the need for stalling all of the mutators.

We observe that the allocation rate is highly architecture dependent, and is the slowest on the SCC. Allocation rate is particularly dependent on memory bandwidth, processor speed and cache behavior. On the SCC, not only is the processor slow (533MHz) but the serial memory bandwidth for our experimental setup is only around 70 MB/s.

7.2 Performance

Next, we analyze the performance of the new local collector design. In order to establish a baseline for the results presented, we have ported our runtime system to utilize the Boehm-Demers-Weiser (BDW) conservative garbage collector [7]. We briefly describe the port of our runtime system utilizing BDW GC.

Although BDW GC is conservative, it can utilize tracing information when provided. Our compiler generates tracing information for all objects, including the stack. However, we provide the tracing information for all object allocations except the stack. Stack objects in our runtime system represent all of the reserved space for a stack, while only a part of the stack is actually used which can grow and shrink as frames are pushed and popped. Since the BDW GC does not allow tracing information of objects to be changed after allocation, we scan stack objects conservatively. BDW uses a mark-sweep algorithm, and we enable parallel marking and thread-local allocations.

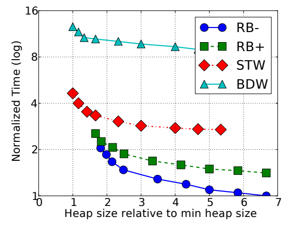
Figure 11a illustrates space-time trade-offs critical for any garbage collector evaluation. STW GC is the baseline stop-the-world collector described in Section 3.2, while RB+ and RB- are local collectors. RB+ is a local collector with read barriers while RB- is our new local collector design without read barriers, exploiting procrastination and cleanliness. We compare the normalized running times of our benchmarks under different garbage collection schemes as we decrease the heap size. For each run of the experiment, we decrease the maximum heap size allowed and report the maximum size of the heap utilized. Thus, we leave it to the collectors to figure out the optimal heap size, within the allowed space. This is essential for the local collectors, since the allocation pattern of each core is usually very different and depends on the structure of the program.

The results presented here were collected on 16 cores. As we decrease overall heap sizes, we see programs under all of the different GC schemes taking longer to run. But RB- exhibits better performance characteristics than its counterparts. We observe that the minimum heap size under which the local collectors would run is greater than the STW and BDW GCs. In the local collectors, since the heap is split across all of the cores, there is more fragmentation. Also, under the current scheme, each local collector is greedy and will try to utilize as much heap as it can in order to reduce the running time (by choosing semi-space collection over mark-compact), without taking into account the heap requirements of other local collectors. Currently, when one of the local cores runs out of memory, we terminate the program. Since we are interested in throughput on scalable architectures where memory is not a bottleneck, we have not optimized the collectors for memory utilization. We believe we can modify our collector for memory constrained environments by allowing local heaps to shrink on demand and switch from semi-space to compacting collection, if other local heaps run out of memory.

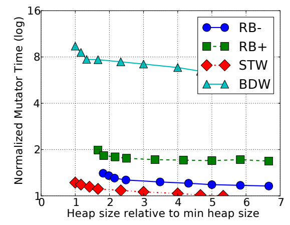
The STW and BDW GCs are much slower than the two local collectors. In order to study the reason behind this slowdown, we separate the mutator time (Figure 11b) and garbage collection time (Figure 11c). We see that STW GC is actually faster than the local collectors in terms of mutator time, since it does not pay the overhead of executing read or write barriers. But, since every collection requires stopping all the mutators and a single collector performs the collection, it executes serially during a GC. Figure 11d shows that roughly 70% of the execution total time for our benchmarks under STW is spent performing GCs, negatively impacting scalability.

Interestingly, we see that programs running under the BDW GC are much slower when compared to other GCs. This is mainly due to allocation costs. Although we enabled thread-local allocations, on 16 cores, approximately 40% of the time was spent on object allocation. While the cost of object allocation for our other collectors only involves bumping the frontier, allocation in BDW GC is significantly more costly, involving scanning through a free list, incurring substantial overhead. Moreover, BDW GC is tuned for languages like C/C++ and Java, where the object lifetimes are longer and allocation rate is lower when compared to functional programming languages.

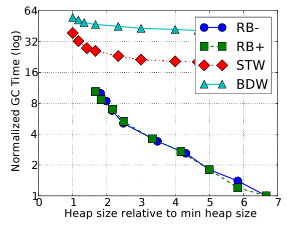
In Figure 11a, at 3X the minimum heap size, RB+, STW and BDW GCs are 32%, 106% and 584% slower than the RB- GC. We observe that there is very little difference between RB+ and RB- in terms of GC time but the mutator time for RB+ is consistently higher than RB- due to read barrier costs. The difference in mutator times is consistent since it is not adversely affected by the increased number of GCs incurred as a result of smaller heap sizes. This also



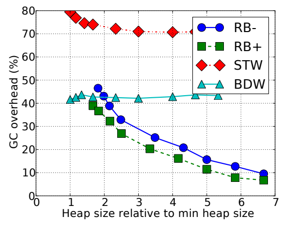
(a) Total time



(b) Mutator time

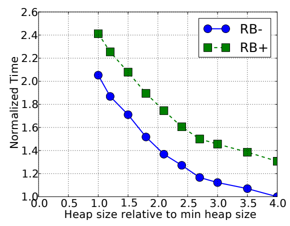


(c) GC time

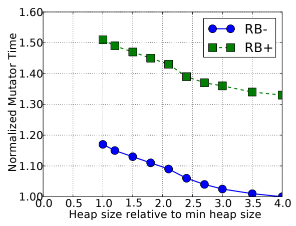


(d) GC overhead

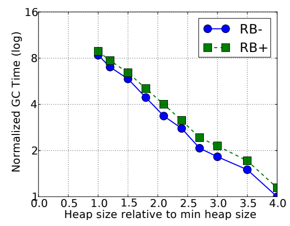
Figure 11: Performance comparison of Stop-the-world (STW), Boehm-Demers-Weiser conservative garbage collector (BDW), local collector with read barriers (RB+), and local collector without read barriers (RB-): Geometric mean for 8 benchmarks running on AMD64 with 16 cores.



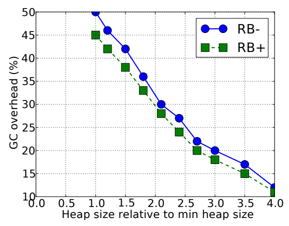
(a) Total time



(b) Mutator time

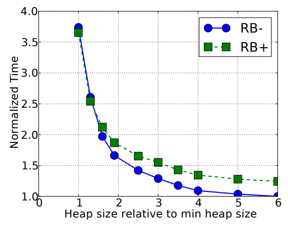


(c) Garbage collection time

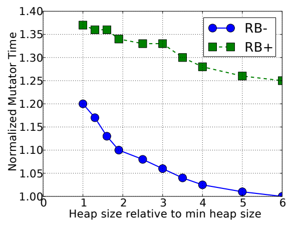


(d) Garbage collection overhead

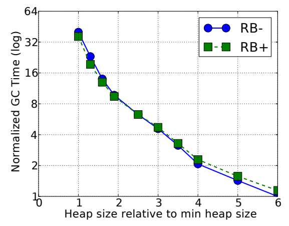
Figure 12: Performance comparison of local collector with read barriers (RB+) and local collector without read barriers (RB-): Geometric mean for 8 benchmarks running on Azul with 846 cores.



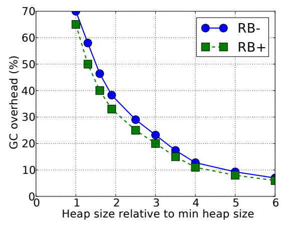
(a) Total time



(b) Mutator time



(c) Garbage collection time



(d) Garbage collection overhead

Figure 13: Performance comparison of local collector with read barriers (RB+) and local collector without read barriers (RB-): Geometric mean for 8 benchmarks running on SCC with 48 cores.

explains why the total running time of RB- approaches RB+ as the heap size is decreased in Figure 11a. With decreasing heap size, the programs spend a larger portion of the time performing GCs, while the mutator time remains consistent. Hence, there is diminishing returns from using RB- as heap size decreases.

Next, we analyze the performance on Azul (see Figure 12). We only consider performance of our local collectors since our AMD results show that the other collectors (STW and BDW) simply do not have favorable scalability characteristics. At 3X the minimum heap size, RB- is 30% faster than RB+.

SCC performance results are presented in Figure 13. At 3X the minimum heap size, RB- is 20% faster than RB+. From the total time graphs, we can see that the programs tend to run much slower as we decrease the heap sizes on SCC. Compared to the fastest running times, the slowest running time for RB- is 2.01X, 2.05X, and 3.74X slower on AMD, Azul, and SCC respectively. This is due to the increased number of shared heap collections, which are more expensive than other architectures as a result of the absence of caching. This is noticeable by a more rapid increase in garbage collection overhead percentages (Figure 13d).

7.3 Impact of cleanliness

Cleanliness information allows the runtime system to avoid preempting threads on a write barrier when the source of an exporting write is clean. In order to study the impact of cleanliness, we removed the reference counting code and cleanliness check from the write barrier; thus, every exporting write results in a thread preemption and stall. The results presented here were taken on the AMD machine with programs running on 16 cores with the benchmark configurations given in Figure 10. The results will be similar on SCC and Azul.

Benchmark	AllPairs	BarnesHut	CountGraphs	GameOfLife	Kclustering	Mandelbrot	Nucleic	Raytrace
RB-	1831	46532	154	38621	25812	132	156	3523
RB- MU-	1831	4092312	192	735543	50323	209	433092	3743
RB- CL-	124232	67156821	50178	5867423	27023911	25491	912349	61198

Figure 14: Number of preemptions on write barrier.

Benchmark	AllPairs	BarnesHut	CountGraphs	GameOfLife	Kclustering	Mandelbrot	Nucleic	Raytrace
RB-	0.08	0.17	0	3.54	0	1.43	0	1.72
RB- MU-	0.08	19.2	0.03	9.47	0.02	2.86	9.37	1.72
RB- CL-	38.55	100	0.18	99.75	21.64	86.22	19.3	24.86

Figure 15: Forced GCs as a percentage of the total number of major GCs.

Figure 14 shows the number of preemptions on write barrier for different local collector configurations. RB- row represents the local collector designs with all of the features enabled; RB- MU- row shows a cleanliness optimization that does not take an object's mutability into consideration in determining cleanliness (using only recorded reference counts instead), and row RB- CL- row represents preemptions incurred when the collector does not use any cleanliness information at all. Without cleanliness, on average, the programs perform substantially more preemptions when encountering a write barrier.

Recall that if all of the threads belonging to a core get preempted on a write barrier, a local major GC is *forced*, which lifts all of the sources of exporting writes, fixes the references to forwarding pointers and unblocks the stalled threads. Hence, an increase in the number of preemptions leads to an increase in the number of local collections.

Figure 15 shows the percentage of local major GCs that were forced compared to the total number of local major GCs. Row RB- CL- shows the percentage of forced GCs if cleanliness information is not used. On average, 49% of local major collection performed is due to forced GCs if cleanliness information is not used, whereas it is less than 1% otherwise. On benchmarks like `BarnesHut`, `GameOfLife` and `Mandelbrot`, where all of the threads tend to operate on a shared global data structure, there are a large number of exporting writes. On such benchmarks almost all local GCs are forced in the absence of cleanliness. This adversely affects the running time of programs.

Figure 16 shows the running time of programs without using cleanliness. On average, programs tend to run 28.2% slower if cleanliness information is ignored. The results show that cleanliness analysis therefore plays a significant role in our GC design.

7.4 Impact of immutability

If the source of an exporting write is immutable, we can make a copy of the object in the shared heap and assign a reference to the new shared heap object to the target. Hence, we can ignore the reference count of such objects. Not all languages may have the ability to distinguish between mutable and immutable objects in the compiler or in the runtime system. Hence, we study the impact of our local collector design with mutability information in mind. To do this, we ignore the test for mutability in the cleanliness check

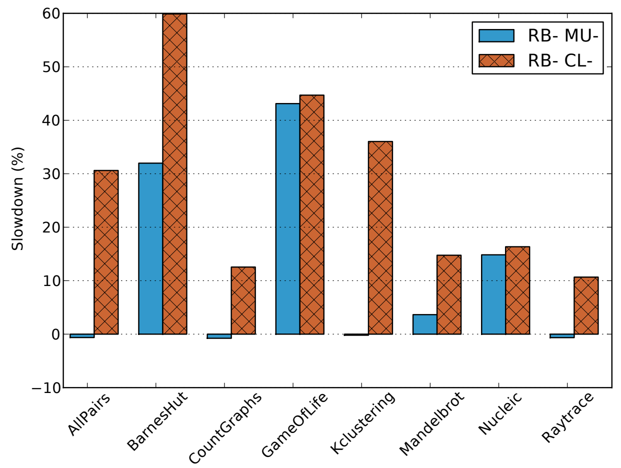


Figure 16: Impact of utilizing object mutability information and cleanliness analysis on the performance of RB- GC.

Benchmark	AllPairs	Barneshut	Countgraphs	GameOfLife	Kclustering	Mandelbrot	Nucleic	Raytrace
% LM clean	5.3	13.4	8.6	23.2	17.6	4.5	13.3	8.2
Avg. session size (bytes)	2908	1580	3612	1344	2318	8723	1264	1123

Figure 17: Impact of heap session: % LM clean represents the fraction of instances when a clean object closure has at least one object with `LOCAL_MANY` references.

(Line 4 in Figure 6) and modify the object lifting code in Figure 8 to treat all objects as mutable.

RB- MU- row in Figure 14 and Figure 15 show the number of write barrier preemptions and the percentage of forced GCs, respectively, if all objects were treated as mutable. For some programs such as `AllPairs`, `CountGraphs`, or `Kclustering`, object mutability does not play a significant factor. For benchmarks where it does, distinguishing between mutable and immutable objects helps avoid inducing preemptions on a write barrier since a copy of the immutable object can be created in the shared heap without the need to repair existing references to the local heap copy.

Figure 16 shows the performance impact of taking object mutability into account. `BarnesHut`, `GameOfLife` and `Nucleic` are slower due to the increased number of forced GCs. Interestingly, `AllPairs`, `CountGraphs`, `Kclustering` and `Raytrace` are marginally faster since they avoid manipulating the `imSet` (Line 14 in Figure 8) and walking immutable objects after the objects are lifted (Lines 25-27 in Figure 8). On average, we see a 11.4% performance impact if mutability information is not utilized for cleanliness.

7.5 Impact of heap session

In order to assess the effectiveness of using heap sessions, we measured the percentage of instances where the source of an exporting write is clean with at least one of the objects in the closure has a `LOCAL_MANY` reference. During such instances, we walk the current heap session to fix any references to forwarded objects. Without using heap sessions, we would have preempted the thread in the write barrier, reducing available concurrency. The results were obtained on the AMD with programs running on 16 cores with the configuration given in Figure 10. The results are presented in Figure 17.

The first row shows the percentage of instances when an object closure is clean and has at least one object with `LOCAL_MANY` references. On average, we see that 12% of clean closures have at least one object with `LOCAL_MANY` references. We also measured the average size of heap sessions when the session is traced as a part of lifting an object closure to the shared heap (Lines 29-31 in Figure 8). The average size of a heap session when it is traced is 2859 bytes, which is less than a page size. These results show that utilizing heap sessions significantly contributes to objects being tagged as clean, and heap sessions are small enough to not introduce significant overheads during tracing.

8. Related Work

Modern garbage collectors rely on read and write barriers for encapsulating operations to be performed when the mutator reads or writes a reference from or to some heap allocated object. The Baker read barrier [5] was the first to use protection and invariants for mutator accesses. While the Baker read barrier is a conditional read barrier, the Brooks read barrier [8] is an unconditional read barrier, where all loads unconditionally forward a pointer in the object header to get to the object. For objects that are not forwarded, this pointer points to the object itself. The Brooks read barrier eliminates branches but increases the size of objects. Trading branches with loads is not a clear optimization as modern processors allow speculation through multiple branches, especially ones that are infrequent.

Over the years, several local collector designs [1, 9, 21, 22] have been proposed for multithreaded programs. Recently, variations of local collector design have been adopted for multithreaded, functional language runtimes like GHC [14] and Manticore [3]. Doligez et al. [9] proposed a local collector design for ML with threads where all mutable objects are allocated directly on the shared heap, and immutable objects are allocated in the local heap. Similar to our technique, whenever local objects are shared between cores, a copy of the immutable object is made in the shared heap. Although this design avoids the need for read and write barriers, allocating all mutable objects, irrespective of their sharing characteristics can lead to poor performance due to increased number of shared collections, and memory access overhead due to NUMA effects and uncached shared memory as in the case of SCC. It is for this reason we do not treat the shared memory as the oldest generation for our local generation collector unlike other designs [9, 14].

Several designs utilize static analysis to determine objects that might potentially escape to other threads [13, 22]. Objects that do not escape are allocated locally, while all others are allocated in the shared heap. The usefulness of such techniques depends greatly on the precision of the analysis, as objects that might potentially be shared are allocated on the shared heap. This is undesirable for architectures like the SCC where shared memory accesses are very expensive compared to local accesses. Compared to these techniques, our design only exports objects that are definitely shared between two or more cores. Our technique is also agnostic to the source language, does not require static analysis, and hence can be implemented as a lightweight runtime technique.

Anderson [1] describes a local collector design (TGC) that triggers a local garbage collection on every exporting write of a mutable object, while immutable objects, that do not have any pointers, are copied to the shared heap. This scheme is a limited form of our cleanliness analysis. In our system, object cleanliness neither solely relies on mutability information, nor is it restricted to objects without pointer fields. Moreover, TGC does not exploit delaying exporting writes to avoid local collections. However, the paper proposes several interesting optimizations that are applicable to our system. In order to avoid frequent mutator pauses on exporting writes, TGC's local collection runs concurrently with the mutator. Though running compaction phase concurrently with the mutator would require read barriers, we can enable concurrent marking to minimize pause times. TGC also proposes watermarking scheme for minimizing stack scanning, which can be utilized in our system to reduce the stack scanning overheads during context switches and exporting writes of clean objects.

Marlow et al. [14] propose exporting only part of the transitive closure to the shared heap, with the idea of minimizing the objects that are globalized. The rest of the closure is exported essentially on demand during the next access from another core. This design mandates the need for a read barrier to test whether the object being accessed resides in the local heap of another core. However, since the target language is Haskell, there is an implicit read barrier on every load, to check whether the thunk has already been evaluated to a value. Since our goal is to eliminate read barriers, we choose to export the transitive closure on an exporting write.

9. Conclusions

The use of read barriers can impose non-trivial overheads in managed languages. In this paper, we consider a design of a runtime system for a thread-aware implementation of Standard ML that completely eliminates the need for read barriers. The design employs a split-heap to allow concurrent local collection, but exploits notions of procrastination and cleanliness to avoid creating forwarding pointers. Procrastination stalls threads about to perform an operation that would otherwise introduce a forwarding pointer, and thus can be used to eliminate read barriers for any exporting write. Cleanliness is an important optimization that helps avoid the cost of stalling by using runtime information to determine when it is safe to copy (rather than move) an object, deferring repair of pointers from the old (local) instance of the object to the new (shared) copy until a later collection. Experimental results on a range of benchmarks and architectural platforms indicate that read barrier elimination contributes to notable performance improvement without significantly complicating the runtime system.

Acknowledgments

We would like to thank our shepherd, Dave Detlefs, and the other anonymous reviewers for their detailed comments and suggestions. This work is supported by the National Science Foundation under grants CCF-0811631 and CNS-0958465, and by gifts from Intel and Samsung Corporation.

References

[1] T. A. Anderson. Optimizations in a Private Nursery-based Garbage Collector. In *ISMM*, pages 21–30, 2010.

[2] A. W. Appel. Simple Generational Garbage Collection and Fast Allocation. *Software Practice and Experience*, 19:171–183, February 1989.

[3] S. Auhagen, L. Bergstrom, M. Fluet, and J. Reppy. Garbage Collection for Multicore NUMA Machines. In *Workshop on Memory Systems Performance and Correctness*, pages 51–57, 2011.

[4] D. F. Bacon, P. Cheng, and V. T. Rajan. A Real-Time Garbage Collector with Low Overhead and Consistent Utilization. In *POPL*, pages 285–298, 2003.

[5] H. G. Baker, Jr. List Processing in Real Time on a Serial Computer. *Communication of the ACM*, 21:280–294, 1978.

[6] S. M. Blackburn and A. L. Hosking. Barriers: Friend or Foe? In *ISMM*, pages 143–151, 2004.

[7] H. Boehm. A Garbage Collector for C and C++, 2012. URL `http://www.hpl.hp.com/personal/Hans_Boehm/gc`.

[8] R. A. Brooks. Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware. In *Lisp and Functional Programming*, pages 256–262, 1984.

[9] D. Doligez and X. Leroy. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *POPL*, pages 113–123, 1993.

[10] L. Gidra, G. Thomas, J. Sopena, and M. Shapiro. Assessing the scalability of garbage collectors on many cores. *SIGOPS Operating Systems Review*, 45(3):15–19, 2012.

[11] P. Hartel, M. Feeley, M. Alt, and L. Augustsson. Benchmarking Implementations of Functional Languages with "Pseudoknot", a Float-Intensive Benchmark. *Journal of Functional Programming*, 6(4):621–655, 1996.

[12] Intel. SCC Platform Overview, 2012. URL `http://communities.intel.com/docs/DOC-5512`.

[13] R. Jones and A. C. King. A Fast Analysis for Thread-Local Garbage Collection with Dynamic Class Loading. In *International Workshop on Source Code Analysis and Manipulation*, pages 129–138, 2005.

[14] S. Marlow and S. Peyton Jones. Multicore Garbage Collection with Local Heaps. In *ISMM*, pages 21–32, 2011.

[15] R. Milner, M. Tofte, and D. Macqueen. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1997.

[16] MLton. The MLton Compiler and Runtime System, 2012. URL `http://www.mlton.org`.

[17] MultiMLton. MLton for Scalable Multicore Architectures, 2012. URL `http://multimlton.cs.purdue.edu`.

[18] J. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 2007.

[19] P. M. Sansom. Dual-Mode Garbage Collection. In *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, pages 283–310, 1991.

[20] F. Siebert. Limits of parallel marking garbage collection. In *ISMM*, pages 21–29, 2008.

[21] G. L. Steele, Jr. Multiprocessing Compactifying Garbage Collection. *Communcations of the ACM*, 18:495–508, September 1975.

[22] B. Steensgaard. Thread-Specific Heaps for Multi-Threaded Programs. In *ISMM*, pages 18–24, 2000.

[23] Streambench. The STREAM Benchmark: Computer Memory Bandwidth, 2012. URL `http://http://www.streambench.org/`.

[24] L. Ziarek, K. Sivaramakrishnan, and S. Jagannathan. Composable Asynchronous Events. In *PLDI*, pages 628–639, 2011.