Dynamic Filtering: Multi-Purpose Architecture Support for Language Runtime Systems

Tim Harris⋆ Saša Tomić† Adrián Cristal† Osman Unsal†

Microsoft Research⋆ BSC-Microsoft Research Center†

tharris@microsoft.com {sasa.tomic, adrian.cristal, osman.unsal}@bsc.es

Abstract

This paper introduces a new abstraction to accelerate the read-barriers and write-barriers used by language runtime systems. We exploit the fact that, dynamically, many barrier executions perform checks but no real work—e.g., in generational garbage collection (GC), frequent checks are needed to detect the creation of inter-generational references, even though such references occur rarely in many workloads. We introduce a form of dynamic filtering that identifies redundant checks by (*i*) recording checks that have recently been executed, and (*ii*) detecting when a barrier is repeating one of these checks. We show how this technique can be applied to a variety of algorithms for GC, transactional memory, and language-based security. By supporting dynamic filtering in the instruction set, we show that the fast-paths of these barriers can be streamlined, reducing the impact on the quality of surrounding code. We show how we accelerate the barriers used for generational GC and transactional memory in the Bartok research compiler. With a 2048-entry filter, dynamic filtering eliminates almost all the overhead of the GC write-barriers. Dynamic filtering eliminates around half the overhead of STM over a non-synchronized baseline—even when used with an STM that is already designed for low overhead, and which employs static analyses to avoid redundant operations.

Categories and Subject Descriptors C.0 [*General*]: Hardware / software interfaces; D.1.3 [*Programming Techniques*]: Concurrent Programming—Parallel programming; D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

General Terms Languages, Performance

1. Introduction

Programming language implementations frequently interpose on the reads and writes that an application makes. For instance, concurrent garbage collectors need to detect conflicts between work by the application and work by the GC. Reference counting GC needs to record how the number of references to an object changes as the heap is manipulated. Runtime systems with thread-private heaps must track when objects in one thread's heap escape to other threads. Implementations of software transactional memory (STM) must perform logging work to track the accesses that a transaction makes, and to detect conflicts between the accesses made by different transactions. Language-based security techniques, such as software fault isolation (SFI), or write integrity testing (WIT), must check that a thread's memory accesses conform to a policy.

Managed runtime systems, such as the JVM or CLR, frequently include different kinds of barriers for different purposes—potentially interposing multiple times on the same memory accesses. Future runtime systems may employ barriers heavily, using more complex barriers for soft-real-time concurrent GC, or to partition the heap into portions that can be managed independently on different clusters of cores.

In this paper we introduce a new abstraction to accelerate a range of these read/write-barriers. We exploit two observations:

- Dynamically, many barriers perform checks but no "real" work. For instance, a write-barrier for generational GC must detect whenever old-to-young references are created. However, such references occur rarely in many programs.
- Many barriers are *self-healing* [15], meaning that after a barrier has executed once on a given set of inputs, then it does not need to execute again. For instance, after a location has been logged as an old-to-young reference, then the same location does not need further checks in the current GC cycle.

Our approach is to accelerate barriers by keeping track, dynamically, of a set of input values on which the barriers have recently been executed—e.g., locations that have recently been logged by a write-barrier for generational GC. The self-healing property means that a barrier can be filtered out if its inputs are already in this set.

We extend the instruction set with an operation, `dyfl`, to perform this kind of dynamic filtering (Section 2). An accelerated barrier implementation uses a `dyfl` instruction to test if the barrier's inputs are already in the set. If so, then the same barrier has already been executed, and no further work is needed. If not, then the barrier executes as normal before updating the set.

Each `dyfl` instruction takes up to two inputs. The inputs are typically addresses, although they are treated as opaque integers without any kind of translation or protection check. From these inputs, it computes a key to probe for in the set: for each input, a mask supplied to `dyfl` can select to use either the complete value, or to quantize it at the granularity of sub-page-size "cards" (512 bytes, as in some card-table-based GC techniques [25]).

This design lets `dyfl` be used with a wide range of different barriers: using two addresses lets us track source-target pairs on reference assignments, and using cards provides a way to exploit spatial locality when a check performed by one barrier is sufficient to remove checks on nearby addresses. Providing this limited amount of input processing means that many barriers' checks can

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.

ASPLOS'10, March 13–17, 2010, Pittsburgh, Pennsylvania, USA.

Copyright © 2010 ACM 978-1-60558-839-1/10/03…$10.00

be expressed as a single dyfl instruction, thereby reducing register pressure and improving the quality of surrounding code.

The set maintained by dyfl is lossy: the implementation can discard any item at any time. These losses can cause a barrier to be repeated un-necessarily, but they cannot cause necessary work to be missed. These semantics are designed to enable a wide range of implementation choices—e.g., over the capacity that underpins the filters, and whether or not this is shared between cores. This form of lossy behavior is the *opposite* sense to Bloom filters [9].

Semantically, dynamic filtering maintains sets on a per-hardware-thread basis. This is essential for use by applications such as STM: barrier work is specific to a given thread, rather than shareable across a whole process. We sketch, in Section 2, how dyfl could be extended to support a mix of per-thread and per-process state.

In Section 3 we show, through a series of case studies, which read/write barriers can benefit from dyfl. Dynamic filtering is most applicable when (*i*) the checks performed by a barrier are complex (e.g., they require registers for temporary values, harming the quality of surrounding code), (*ii*) the barrier performs additional, occasional, work (e.g., logging) rather than changing every memory access (e.g., indirecting all reads via a forwarding pointer).

We evaluate dyfl in Section 4. We use an x86 simulator with a simple model of a cache hierarchy, and study the use of dyfl in generational GC and STM. Generally, we model a dyfl implementation that uses a table of 2048 entries, with just 1-way, 2-way, or 4-way associativity on lookups based on a hash of the key (rather than needing a CAM). Dynamic filtering eliminates almost all the overhead of the GC write-barriers. Dynamic filtering eliminates around half the overhead of STM over a non-synchronized baseline (all our results use an STM that is designed for low overhead, and employs static analyses to avoid redundant operations).

We discuss related work in detail in Section 5. At a high level, there are two ways in which our design for dyfl generalizes earlier work (e.g., Saha *et al.*'s [34]) on eliding read/write barriers by filtering repetitions. First, dyfl allows repetition checks to use pairs of addresses, rather than single addresses. Second, dyfl allows addresses to be quantized, rather than treating the entire address as significant. As our evaluation shows, both of these design choices are necessary to get a high hit rate in the workloads that we have studied.

Throughout this paper, we make design choices that avoid additional complexity beyond the basic per-thread two-input lossy form of dyfl. For instance, we maintain fixed-size sets of filter entries (rather than trying to virtualize the sets), we use a fixed card size (rather than controlling it dynamically), we expose only a simple present/absent result when making a filter test (rather than storing application-supplied meta-data), and we do not examine integration between the filter mechanism and the caches (for instance, adding or removing filter entries in response to coherence traffic). One could imagine relaxing any of these decisions to explore the trade-off between supporting additional applications for dyfl, versus additional design complexity. Equally, one could combine per-core filtering via dyfl with mechanisms for cross-core conflict detection such as SoftSig [37]. We leave this exploration for future work.

2. Dynamic Filtering

In this section we introduce dyfl using a simple write-barrier for generational GC. In C-like pseudo-code, the original barrier is:

```
void writeBarrier(void **addr, void *tgt) {
  if (inOldGen(addr) && inYoungGen(tgt)) {  // T1
    log(addr);                               // L1
} }
```

The barrier needs to examine each *addr=tgt assignment. It logs addr if that address contains an old-to-young reference. In practice many other forms of write-barrier exist (we discuss examples in Section 3), but this simple case serves as an illustration. Typically, an implementation would inline the check (T1) at each assignment, and keep the logging work (L1) out-of-line. Blackburn *et al.* provide thorough studies of implementation techniques [7, 8].

We refer to the check T1 as the "full check" that the barrier performs (to distinguish it from the "accelerated check" that dynamic filtering will add). We say that the full check "succeeds" when it determines that there is no further work to do, and the "hit rate" of a check is the fraction of time it succeeds. We refer to the work L1 as the "slow-path work" of the barrier. When we refer to the "same barrier" we mean identical input parameters, rather than a specific barrier in the program code.

In this example, a write-barrier is unnecessary if either of two conditions holds:

1. If the (addr,tgt) pair has already passed the full check—e.g., due to an earlier execution of the barrier.
2. If addr has already been logged by the slow-path work—e.g., due to a previous update to the same address.

Both of these conditions are expressed in terms of the *repetition* of a full check, rather than in terms of the details of how the full check itself is implemented—a repetition of the barrier can be removed, no matter how the internals of the inOldGen and inYoungGen functions are organized.

Dynamic filtering provides a mechanism for keeping track of the full checks that have already been made, and for testing for repetitions of these checks. The example barrier can be rewritten:

```
void writeBarrierDyfl(void **addr, void *tgt) {
  if ((!dyfl_card_pair(addr, tgt, 0x1)) &&      // A1
      (!dyfl_addr(addr, 0x2))) {                // A2
    if (inOldGen(addr) && inYoungGen(tgt)) {    // T1
      dyfl_set_addr(addr, 0x2);                 // S2
      log(addr);                                // L1
    } else {
      dyfl_set_card_pair(addr, tgt, 0x1);       // S1
} } }
```

This illustrates two forms of dyfl. The first, dyfl_card_pair, tests if a full check has already been done on the given (addr,tgt) pair. The dyfl implementation keeps track of pairs that have been checked, and the *tag* 0x1 distinguishes this use of dyfl from other uses (e.g., from other barriers used by the same thread).

The fact this is a *card-pair* check means that the inputs are both quantized to a 512-byte card granularity. Assuming that objects in the same card are always in the same generation, this quantization lets us exploit spatial locality: checks done for one pair of addresses can be reused for subsequent addresses on the same cards. (We examine the sensitivity of our results to the specific choice of card size in Section 4.)

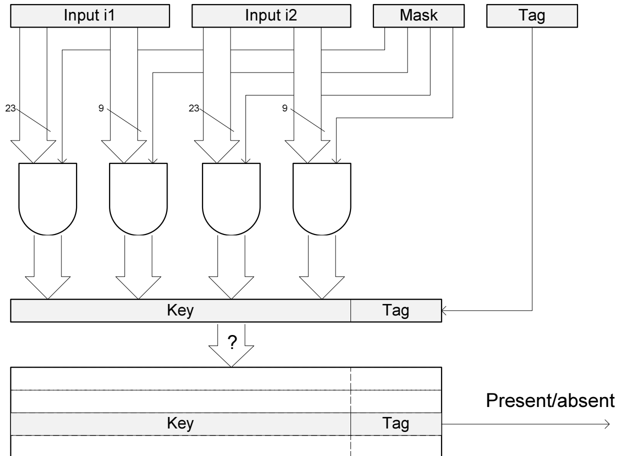
The second operation, dyfl_addr, is a simple single-address check on the input address. In this case the entire address is significant, rather than being quantized. As before, the tag 0x2 distinguishes this check from other uses of dyfl.

If either of these checks succeeds then the full-check T1 can be skipped. If both of these checks fail then the barrier executes as normal, performing the check T1, and possibly performing the slow-path work L1. The dyfl_set operations are used to update the filter state that is queried by dyfl. S1 records that a given (addr,tgt) card-pair has been checked as valid. S2 records that a given addr has been logged.

For this barrier, it makes sense to check A1 before A2. This is because most updates to the heap do not create old-to-young references and so, once a card-pair has been checked, future executions will pass A1. The single-address repetition check A2 serves only to filter out repeated updates to the same old-generation location.

```
dyfl(i1, i2, mask, tag)       // Test dynamic filter
dyfl_set(i1, i2, mask, tag)   // Set dynamic filter
dyfl_clear(i1, i2, mask, tag) // Clear specific entry
dyfl_clear(tag)               // Clear all with tag
```

(a) `dyfl` and associated operations.



(b) Implementation sketch.

Figure 1. Dynamic filtering operations and implementation sketch. The `mask` selects which portions of `i1` and `i2` are significant. The result is combined with the `tag` and tested in the table.

2.1 Dynamic Filtering in the ISA

In this section we discuss detailed design choices, and how we express `dyfl` in the concrete x86 instruction set architecture (ISA) targeted by the compiler we use.

We generalize `dyfl_card_pair` and `dyfl_addr` by a single `dyfl` operation that takes two input values (`i1`, `i2`), and a 4-bit `mask` that selects which portions of the inputs are significant. The mask allows each input to be taken in its entirety, or to be quantized to a card granularity. Figure 1 shows the complete set of operations provided. Several masks recur, and so we give distinct names to operations using them:

- `dyfl_addr(i1)` —using just `i1` and ignoring `i2`.
- `dyfl_card_addr(i1)` —selecting just the card portion of `i1`, and ignoring the lower bits.
- `dyfl_addr_pair(i1,i2)` —selecting the entirety of both inputs.
- `dyfl_card_pair(i1,i2)` —selecting the card portion of both inputs.

For consistency with existing x86 instructions, and consequently simplicity in the compiler, we encode `dyfl` using an ordinary two-operand format, with one operand an effective address (EA) providing `i1`, and the second operand a constant encoding the tag and mask. When `i2` is needed, it is always taken from a fixed register (`edx`), as with a few other x86 instructions.

We signal filter hits/misses via the flags register. (We originally considered using a user-mode trap to a pre-registered handler. However, that approach prevents the handler from being specialized to a particular occurrence of a barrier in the program—e.g., depending on which registers are available for scratch use, or where the inputs to the barrier are stored. In addition, using the flags register lets us have different handlers for different operations using the same tag. We exploit this when using `dyfl` with STM: we can use a single tag to track which locations have already been read, but employ a different miss-handler on indirect accesses which must map interior pointers to object headers.)

Example. Returning to our example barrier for generational GC, the x86 fast-path implementation is:

```
     // Assuming the store will be *ecx = edx
    dyfl_card_pair [ecx] <- 0x1 // A1
    jz L0
    dyfl_addr [ecx] <- 0x2      // A2
    jnz L1
L0: mov [ecx] <- edx
    ...
L1: // Out-of-line full-check
    jmp L0
```

Ultimately, for `dyfl` to speed up a program, there must be a sufficiently high filter hit rate for the work saved on hits to outweigh the additional tests and book-keeping on misses. In some examples – e.g., the STM in Section 3.2 – these savings come both from the tests themselves, and from improved quality in code around the barrier. Of course, the `dyfl` tests must also be faster than the barrier's full test.

Tag assignment. In our prototype, we assume that there are 16 tags available. Our choice is somewhat arbitrary: it provides more than enough tags for our examples, while letting us pack the mask and tag into a single 8-bit immediate constant (4 bits for the tag and 4 bits for the mask).

In our current implementation, tag numbers occur directly in the native code generated by our compiler, and we assume that the assignment of tags for different purposes is something that the language runtime system's designer co-ordinates. This is much like the overall structure of the virtual address space, the use of x86 segment registers, or the choice of calling convention. Ordinary techniques could combine modules with different conventions; e.g., rewriting tags at load-time.

Sharing. We envisage per-core management of `dyfl` filters. Where multiple hardware threads share a core, the tag could be extended implicitly to distinguish each thread. This is necessary in applications where barrier work from different threads is independent—e.g., thread-private heaps (Section 3.1), or STM (Section 3.2). Alternatively, sharing could be placed under software control by adding a mapping from *logical tags* used by software to *physical tags* stored in the table. Different physical tags would be used when sharing between threads must be prohibited (e.g., STM). Common physical tags would be used when sharing is useful (e.g., generational GC). Of course, threads running code from different address spaces would need to remain distinct.

Implementation. Several implementations are possible, e.g., associating key-tag pairs with cache lines that relate to addresses in the key, or using a separate structure independently from the cache. There are clearly many trade-offs here, and we focus in this paper on the use and possible performance of different kinds of filtering.

Our general approach is motivated by allowing `dyfl` to be implemented independently from the caches, as sketched in Figure 1. Yen *et al.* give strong arguments for this [45]. First, cache tags and arrays are performance-critical; modifications that appear simple may be difficult to implement without a performance penalty. Second, the desire to support multi-threading within processors may require any additional storage to be replicated many times over.

3. Using Dynamic Filtering

In this section we examine a series of barriers, and discuss whether or not `dyfl` could be used to avoid repetition of the checks that they use. The examples in Section 3.1 are from heap management,

the examples in Section 3.2 are from STM, and the examples in Section 3.3 are from language-based security. This is not meant as a complete survey of all existing read/write-barriers. Rather, it provides some confidence that dyfl applies more broadly than the two cases we evaluate in detail on our simulator (Section 4).

3.1 Heap Management and Garbage Collection

Classical generational GC. As our running example showed, dyfl can be used on each assignment *addr=tgt (*i*) to avoid repeated tests of the same (addr,tgt) pair, and (*ii*) to avoid repeated tests on the same addr once it has been logged.

We examine the performance of this in the Bartok runtime system in Section 4. In that setting, a process-wide table records which generation occupies each page. The GC's barrier implementation requires registers for the table base and for temporary values during lookups. Much of the cost of software write-barriers comes from this activity and the register pressure that it adds.

The design of Bartok's generational write-barrier is more complex than other current designs, e.g., those in the Jikes Research Virtual Machine [4]. Primarily, this is because Bartok's runtime system is designed to allow independent applications to co-exist in a single virtual address space. This is used in the Singularity research operating system [23] in which all processes operate over the same physical memory without hardware isolation between them.

Blackburn *et al.* describe the slot-based and object-based write barriers in the MMTk toolkit used by Jikes [8]. MMTk's slot-based barrier determines old-to-young references by comparing the source and target addresses, placing the young generation at the top of the process' virtual address space, and computing the boundary between the generations by a series of shifts:

```
void writeBarrierMMTkPtr(void **source, void *tgt) {
  if (source < ((tgt>>HEAP_K)<<HEAP_K)) {
    log(source);
} }
```

MMTk's object-based barrier records complete objects rather than individual fields. By working at an object granularity, it can use a per-object header flag to record whether or not a given object has already been logged:

```
void writeBarrierMMTkObj(Object source) {
  int statusWord = GetObjectStatusWord(source);
  if ((statusWord & OBJECT_BARRIER_MASK) != 0) {
    logObject(source);
} }
```

The GetObjectStatusWord operation loads a status word from the header of the object, and so the complete fast-path test is extremely straightforward.

Dynamic filtering is typically not effective for these barriers in Jikes (we experimented using modern x86 hardware, comparing the execution time of the mutator for DaCapo benchmarks with, and without, the write-barriers, on a machine with sufficient memory to avoid collection work). As with Blackburn *et al.*'s studies [7, 8], the inclusion of barriers accounted for only a few percent of execution time. The MMTk barriers' full checks are much simpler than those used by Bartok—in terms of their behavior, and register pressure.

However, as we show below, other GC algorithms are more complex than classical generational GC, and the table-based lookups performed by Bartok are common with other applications—e.g., implementations of thread-local heaps with each thread's local data placed on a separate set of pages.

Concurrent mark-sweep (CMS) GC. CMS collectors typically use a write barrier that records information about updates made by the mutator during the marking phase of a GC cycle. For a store *addr=tgt, a Steele-style barrier records addr. A Dijkstra-style barrier records tgt. A Yuasa-style barrier records the overwritten value. Vechev and Bacon's paper summarizes these approaches and identifies conditions for eliminating barriers [40].

Any of these barriers can be elided if the log entry has already been recorded; dyfl_addr can be used in each case. However, there are a two further points to consider. First, some implementations of Dijkstra-style and Yuasa-style barriers require explicit tests for NULL references—e.g., to prevent an access violation if logging is done by setting a bit in an object's header:

```
void dijkstraBarrier(void **addr, void *tgt) {
  if (tgt != null) {
    if (!alreadyMarked(tgt)) {
      mark(tgt);
} } }
```

This example motivates the decision for dyfl to treat addresses as ordinary scalar inputs, without performing access checks on them. This decision allows filtering to be done *before* the NULL test:

```
void dijkstraBarrier(void **addr, void *tgt) {
  if (!dyfl_addr(tgt, 0x1)) {
    if (tgt != null) {
      ...
    } }
    dyfl_set_addr(tgt, 0x1);
} }
```

The second consideration is that, as with generational GC, concurrent GC can use a *card-table* rather than precise logs of objects or addresses. This allows a dyfl_card_addr operation to be used, assuming that the GC's card size is not smaller than dyfl's. Card-based filtering improves the hit rate if there are multiple barriers on locations in the same dyfl card. Integration with card-table GC techniques motivates the use of sub-page-sized cards in dyfl.

Similar uses of dyfl are possible for Levanoni and Petrank's concurrent reference counting algorithm [27], and Azatchi *et al.*'s CMS collector [5].

Kermany and Petrank's compacting GC. Kermany and Petrank's "Compressor" algorithm is a concurrent, incremental, parallel compacting garbage collector [26]. The compaction phase moves the reachable objects into a contiguous region of memory. Mutator threads run concurrently with compaction, and so synchronization is required to handle accesses to objects being moved.

A process's heap appears (broadly speaking) to be similar to a two-space copying collector. However, Compressor uses virtual memory hardware to take pages out of the from-space (once all of the objects on them have been removed), and to reassign them to the to-space. This is done incrementally; initially all pages in to-space are inaccessible, and so accesses trigger violations. Pages in to-space are incrementally constructed in response to these access violations.

Dynamic filtering can be used in place of virtual memory page protection checks, using a dyfl_card_addr check on each location before it is accessed, followed on a miss by a software check of whether or not the location is in to-space.

There is a complicated trade-off in performance between an implementation using dynamic filtering, and an implementation using off-the-shelf page protection hardware. First, dynamic filtering makes the access checks via explicit dyfl instructions, rather than implicit on memory accesses. However, the cost of handling a filter miss is substantially less than an access violation. Second, using dyfl would avoid the need to manipulate page protection settings; such hardware is widely used in research prototypes, but there is reluctance to use it in production settings. Finally, decoupling Compressor's access checks from page-level memory protection can enable the use of larger page sizes (e.g., 2MB pages rather than 4KB).

Pizlo* et al.*'s concurrent copying collectors. Pizlo *et al.* describe a number of concurrent real-time copying collectors [32].

The "Clover" collector relies on the mutator recognizing a special marker value (α) that is placed in a location when it is being used by the collector. A read-barrier performs slow-path work if it sees α: the barrier must locate the object in to-space, and return the value present there. A write-barrier must wait if it is trying to store α while the GC is moving objects. Otherwise, the write-barrier uses CAS when writing to from-space (to prevent races with the collector), or it uses an ordinary store when writing to to-space.

Dynamic filtering could be used in various ways here, e.g., to check that a value read is not equal to α (so that the value may be returned immediately), and to check that a location being written is in to-space (so that the location may be updated directly).

It is not clear whether dynamic filtering can be used with Pizlo *et al.*'s other two concurrent real-time copying collectors. The "Chicken" and "Stopless" collectors both involve extensive barriers during some phases of the GC's work. These do not simply perform logging work or occasional redirection to a different address. For example, "Chicken" involves indirection through an object's header to find the current copy of an object (before/after copying).

Doligez and Leroy's thread-local heaps. Doligez and Leroy's implementation of thread-local heaps for ML [18] maintains an invariant that there are no references from shared data to thread-private data (i.e., it forbids references pointing from shared data into a thread's stack, or pointing into a thread's local heap). This separation lets each thread perform local collection of its private heap.

This invariant can be preserved by a write-barrier. For each store `*addr=tgt`, the barrier checks that if `addr` is a location in the shared heap then `tgt` is also a shared location. If `tgt` is not shared then the object at `tgt` is promoted to the shared heap, and a reference to the new location is written accordingly. At the next GC, information kept at the old location of `tgt` is used to fix up any other references to it (so that `tgt` does not get permanently duplicated). For immutable objects, the temporary distinction between the old copy of `tgt` and the new copy is not visible to the program. Mutable objects are rare in ML and are always allocated in the shared heap.

```
void writeBarrierTLHeap(void **addr, void *tgt) {
  if (inSharedHeap(addr) && !inSharedHeap(tgt)) { // T1
    new_tgt = promote(tgt);
    *addr = new_tgt;
  } else {
    *addr = tgt;
} }
```

Dynamic filtering can be used to record pairs of addresses that have passed test `T1`. If the heap is organized at a card granularity (or coarser) then `dyfl_card_pair` can be used to exploit spatial locality.

Unlike the previous write-barriers, the value written to `addr` is dependent on whether or not the test `T1` succeeds. Consequently, at the level of assembly language, usage is more complex:

```
     // Assuming the store will be *(ecx+4) = edx
    dyfl_card_pair [ecx+4] <- 0x1
    jnz L2
L0: mov [ecx+4] <- edx
L1: ...

L2: // Filter miss handler
```

The filter miss handler performs the full check from the function `writeBarrierTLHeap`, branching back to `L0` if the test succeeds, and branching back to `L1` if the test fails.

An alternative implementation of thread-local heaps may use a read barrier that checks that the pointers followed by a thread refer to the thread's own local data, or to data in the shared heap. If a thread encounters a pointer into another thread's heap then it synchronizes with the object's owner to request that the object be promoted into the shared heap. Dynamic filtering can be used to track which cards have been tested as valid for the current thread to access. The slow-path work could use an ownership table similar to that used to identify generations in the Bartok runtime system.

3.2 Transactional Memory

Transactional memory implementations track the reads and writes made by transactions, and ensure that if concurrent transactions conflict then at most one of them will be committed.

STM with eager updates. Several STM implementations use *eager updates*, meaning that a transaction updates memory directly, while keeping an undo log to roll back its effects if a conflict is detected [20, 33]. Such designs typically use optimistic concurrency control for reads, and locking for writes. When a transaction reads from a location, the STM records meta-data to a thread-private log, and then re-checks this log for conflicts at the end of the transaction. Concurrency control is performed on a per-object basis, and so the same object's meta-data can be accessed repeatedly—e.g., if different elements of the same array are accessed.

Dynamic filtering can be used to check whether or not a location has already been accessed in the current transaction: if it has been accessed, then additional concurrency control work can be avoided. We have prototyped the use of dynamic filtering with Bartok-STM, and examine its performance in Section 4.

STM with deferred updates. Rather than making eager updates, some STMs use *deferred*, or *lazy* updates. These STMs keep a "redo" log of tentative changes that must be written to the heap when a transaction commits.

Dynamic filtering is less applicable to STMs with deferred updates than to STMs with eager updates. This is because the slow-path work of the barriers is needed on most accesses (to make sure that the redo log is consulted on reads, and updated on writes), rather than being performed before the first access whereupon memory can be used directly.

It would be possible to use dynamic filtering to track which locations have *not* been written within the current transaction: a filter entry would indicate that a check has already been done that a location is not in the redo log. Such a read could access memory directly. A subsequent write would clear the filter entry.

3.3 Language-Based Security

In this section we consider software security techniques, e.g., for defending native code from buffer-overflow problems.

Control flow integrity (CFI). CFI checks that a program's dynamic control flow is consistent with a statically-computed safe control flow graph [1]. A static analysis determines the possible targets for each control flow instruction, assigning markers to control flow instructions and to their targets such that all possible targets of a given instruction are assigned the same marker—e.g., all possible branch targets from instruction `B1` might be assigned marker `0x12345`. Dynamically, a control flow instruction is preceded by a check that its target has the marker that is expected. Budiu *et al.* investigated architecture support for CFI via a special form of branch instruction that encodes the expected marker value [10].

Dynamic filtering can be used in one of two ways. First, filters could record target-marker associations. A branch at `B1` would check whether its particular target address is already associated with marker `0x12345`. If so, then the branch has already been checked. If not, then the filter miss handler would check whether or not the target has the marker that is required.

Alternatively, filters could record valid source-target address pairs. A branch at `B1` would then check whether the target address

Total allocation / MB	Dynamic filtering operations GC	STM	Usage of atomic blocks # blocks	Log size	% exec.
Crafty, chess program translated to C#:					
5.1	161 699	0	0	0	0
204.1	3 599 112	0	0	0	0
Delaunay triangulation following Scott's description [36]:					
24.1	6 937 117	0	0	0	0
833.1	81 821 274	0	0	0	0
Genome, sequencing benchmark from STAMP, translated to C#:					
29.3	738 194	3 095 340	74 283	17	59
1587.1	8 893 286	31 081 914	752 406	17	56
Go, the commonly seen Go playing program:					
8.5	5 082	0	0	0	0
714.5	182 775	0	0	0	0
JBBAtomic, JBB ported to C#, with a fixed number of tx, each in an atomic block:					
15.4	853 968	3 400 174	2 000	407	82
523.1	38 715 672	217 039 332	100 000	600	79
Labyrinth, maze routing benchmark from STAMP, translated to C#:					
0.3	240	5 765 360	130	10 940	89
1.8	1 229	91 737 215	514	44 091	99
MaxFlow, max-flow algorithm based on preflow-push:					
4.8	1 175 084	5 728 503	1 158 724	5	77
322.8	81 937 962	426 208 098	81 378 071	5	86
Othello, the commonly seen Othello program:					
0.2	260	0	0	0	0
15.0	13 914	0	0	0	0
SatSolver, SAT satisfiability program:					
6.0	282 751	0	0	0	0
181.4	7 786 473	0	0	0	0
Vacation, travel reservation system model from STAMP, translated to C#:					
4.1	167 871	4 880 521	40 000	50	63
83.3	3 594 324	113 875 772	819 430	57	68
XLisp, the commonly seen LISP implementation:					
6.4	5 633 942	0	0	0	0
235.4	168 249 042	0	0	0	0

Figure 2. Benchmark characteristics. We list the statistics for the simulation workload (first line) and normal workload (second line).

is already associated with `B1`. If so, then the branch is valid. If not, then a full check is needed. This alternative approach may permit more precise checks than using a single marker for all targets—e.g., a separate lookup structure could encode the exact source-target address pairs to allow. Whether or not this is worthwhile will depend on the control-flow attacks to be prevented (increased precision may detect more attacks), and on the hit rate achieved by the filter (a high hit rate reduces the impact of a more complex, precise lookup on misses).

CFI motivates supporting `dyfl` with two inputs, both at full-address granularity, rather than just card-pairs.

XFI. XFI extends CFI with checks on data accesses [38]. The idea is to constrain which memory regions a given module is permitted to write into. This can be implemented as checks before each store, with fast-paths used for common cases (e.g., a module with a single contiguous range). Static analyses can remove redundant checks (e.g., before a series of stores to nearby locations). Budiu *et al.* investigated architectural support for XFI [10] via a form of access-check that compares an address against a pair of bounds.

We can use dynamic filtering based on address-pairs that record which instructions have been tested for access to which memory locations. We can use card-level associations for any complete cards that the range covers. Different tags would distinguish different kinds of access (read, write, execute).

Data flow integrity (DFI) and write integrity testing (WIT). DFI [11] and WIT [3] are both based on instrumenting code with dynamic checks that a given data access is permitted according to the results of a static analysis of a program's correct behavior.

DFI involves instrumenting loads and stores. A store updates a shared table that associates an ID with each memory location. A load checks that a location's current ID is in a set of IDs permitted by the static analysis. Some special cases may be identified by dynamic filtering, but it is not clear that these will be common—e.g., a store's update to the table may be elided if the table in memory is already known to contain a given value.

WIT uses a global "color table" that is initialized when memory is allocated, and checked on every store. Each store instruction may only write to memory of a specific color. As with DFI, repeated initializations to the same color may be elided.

With WIT, a check on a store can be elided if the table must already hold the required value. In principle this could be done by tracking address-color pairs. However, in multi-threaded programs, such filter entries would need to be cleared whenever `addr` is updated, including updates on other processors. Some kind of cache integration might be possible, but it would seem to complicate the semantics and possible implementations.

3.4 Discussion

These examples illustrate some general principles of where dynamic filtering may be useful, and where it is not.

In the successful examples, the structure of the barrier is either (*i*) additional slow-path work before a normal access (e.g., logging concurrency control information before an STM's first access to a location), or (*ii*) alternative occasional slow-path work in place of a normal access (e.g., reading a from-space copy of an object via a level of indirection, rather than reading a to-space copy directly).

In the unsuccessful examples, the structure of the barrier involves work on most memory accesses, with few cases in which the underlying location may be accessed directly.

4. Evaluation

In this section we evaluate the use of `dyfl` in the implementation of barriers for generational garbage collection and for STM. Our work is based on Bartok, an optimizing ahead-of-time research compiler and runtime system [2, 20, 32].

Section 4.1 describes the workloads that we run. Section 4.2 describes the simulator that we built to model `dyfl`. Section 4.3 evaluates the use of `dyfl` for generational GC. Section 4.4 evaluates the use of `dyfl` in the STM implementation. Section 4.5 evaluates both uses of `dyfl` in the same program. Finally, Section 4.6 evaluates the sensitivity of the results to the particular parameters we use in the simulated hardware.

4.1 Workloads

Figure 2 shows the benchmarks we use. These are a combination of ones used in earlier work on GC and STM with Bartok. The majority of the STM workloads come from the publicly-available STAMP benchmark suite [29] and from Scott *et al.*'s implementation of Delaunay triangulation using transactions [36]. In each case, we converted these to C#, following the structure of the original C/C++ implementations (primarily, replacing `struct` definitions with classes, and grouping related functions into methods on those classes). In previous work, we reported that, given Bartok's whole-program optimization, the C# versions of the STM benchmarks perform broadly similar to the original versions of these programs [2].

For each program we use two differently-sized workloads. On real hardware, the smaller workloads each take around 100ms to run and require only a few MB heap allocation. We use these workloads for simulated results. The larger workloads take several seconds to run. We discuss, throughout this section, how we validate that behavior seen during the shorter runs is representative of the larger ones.

4.2 Simulator

Most of our evaluation is based on an x86 simulator. The simulator hosts a single multi-threaded user-mode process, emulating

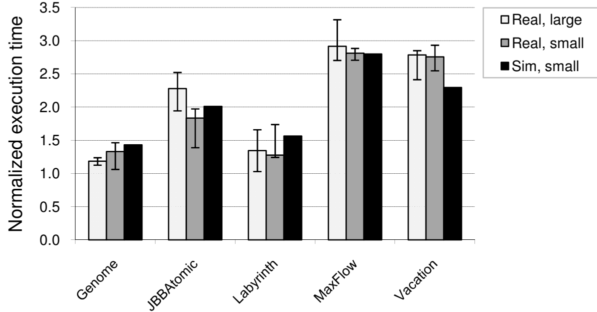


Figure 3. Validation, comparing STM performance on different platforms normalized against unsynchronized performance.

instructions from each thread, and scheduling them according to a timing model. We use a simple timing model which includes a configurable cache-hierarchy: each instruction takes 1 cycle, plus the number of cycles spent on any memory accesses that it performs. All of our results are presented using this timing model (our experimental harness also gathers results in terms of instruction counts, but instruction counts show dyfl in an artificially favorable light because many of the software barriers involve several simple arithmetic operations, e.g., shifting and masking).

We configure the simulator to model a 32KB L1-cache with 4-way associativity, over a 4MB L2-cache with 16-way associativity. The line size is 64 bytes. An L1 hit takes 2 cycles, an L2 hit takes 16 cycles, and a main-memory access takes 200 cycles. Both caches are write back. The parameters are broadly based on current Intel Core 2 Duo processors.

We assume that the dyfl implementation is separate from the caches (in future work, it may be worthwhile to investigate more complicated designs which combine filter information with related cache entries).

Unless otherwise stated, we model dyfl implementations using a 2048-entry filter. To examine the effect of associativity, we configure this as 2048x1-entry sets, 1024x2-entry sets, and 512x4-entry sets. Our model assumes that all dyfl operations take 2 cycles, given that the filter lookup is comparable to a lookup on a cache tag array with a similar size and associativity.

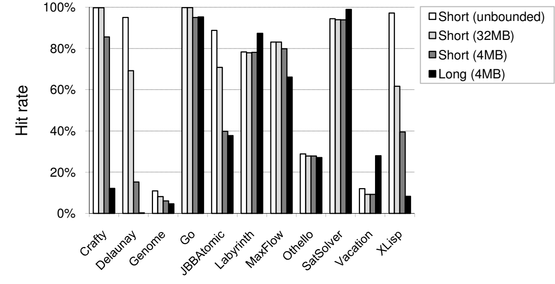
The mapping of key-tag pairs to sets depends on all bits of the key and all bits of the tag. Preliminary results showed that this was essential for low levels of associativity to be effective—otherwise, conflicts occur when multiple tags are associated with the same key, or when the mask leaves portions of the key 0. All our results use LRU replacement of entries within each set (irrespective of the tag involved). We have also collected all our results using random replacement; there is no appreciable difference for the workloads we have studied.

Within the instruction set, we encode a dyfl operation as:

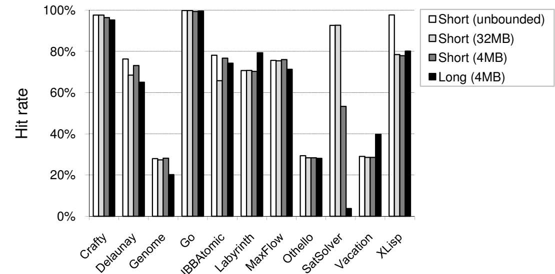
```
ud2                  ; Raise undefined-opcode exception
mov [EA] <- tag_mask ; Store immediate constant to EA
```

The simulator recognizes this sequence and executes it directly as a dyfl operation. For faster turn-around time, during development, we run the program directly and use an in-process handler to respond to the undefined-opcode exception. The handler decodes the dyfl operation and models its effect.

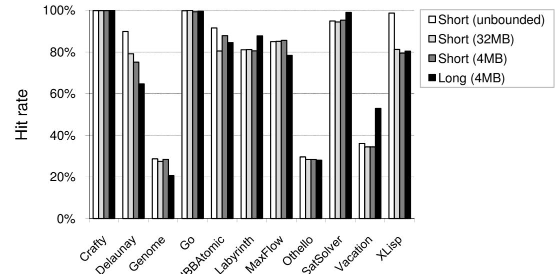
To gain confidence in the simulator's results, we compared performance metrics from a series of executions of the benchmarks. We examined (*i*) executions with large input sets, running on real hardware, (*ii*) executions with simulation input sets, running on real hardware, and (*iii*) executions with simulation input sets, running on simulated hardware. In each case we look at the *ratio* of execution time using STM to the underlying sequential execution of the program without synchronization. The comparison between (*i*) and (*ii*) illustrates whether or not the use of a smaller workload effects our results (e.g., because the heap size is reduced), and the comparison between (*ii*) and (*iii*) illustrates whether or not the simulator's results are faithful to reality.



(a) One-address repetition check (dyfl_addr)



(b) Two-card check (dyfl_card_pair)



(c) Two-card check, followed by one-address repetition check

Figure 4. Generational write-barrier hit rates.

Figure 3 shows these results. For the real runs we collect the median and min/max over 7 runs. For the simulated runs, the results are stable from one run to the next and so we omit error bars (we vary random seeds used by the language runtime system, and the base address of the heap). The simulation results are reasonably close to the timed results, even though the simulator uses a simplistic in-order timing model. These workloads perform large numbers of memory accesses in the STM implementation, and so the major determinant of performance is how these accesses interact with the caches. We would need a more detailed model of the processor core for CPU-bound computational workloads.

4.3 Generational GC with Dynamic Filtering

We employ dynamic filtering in Bartok's generational collector following the design in Section 2. We examine three different kinds of filtering. First, we use only single-address dyfl_addr repetition filtering of addresses being accessed (adding an entry to the filter when an address is logged, or when it is checked to be in the young generation). Second, we use dyfl_card_pair filtering

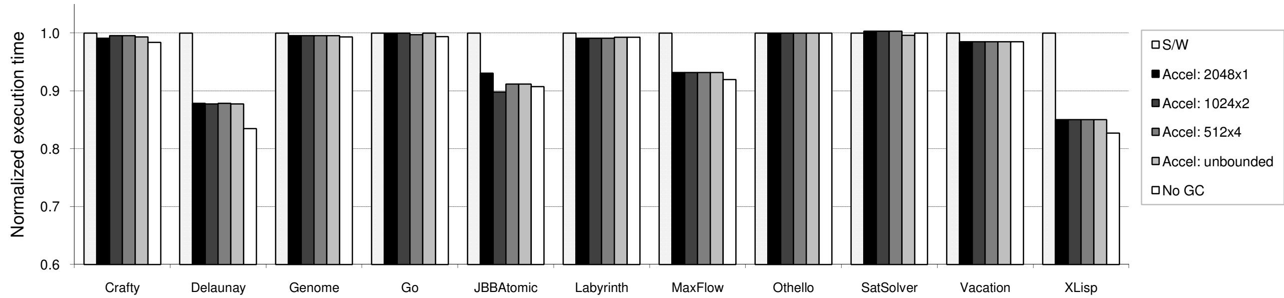


Figure 5. GC acceleration performance, normalized against the original software implementation. The right-most bar in each cluster shows the best possible case, with all write-barriers removed. Accelerated implementations lie between these extremes.

of source-target pairs that have been checked to be valid. Finally, we combine these, as with our pseudo-code in Section 2.

Hit rate. First, we examine the hit rate achieved by the different forms of filtering (Figure 4). The purpose of this is twofold: (*i*) to see whether or not the more complex `dyfl_card_pair` filtering provides a quantitative advantage over the simpler `dyfl_addr` check, and (*ii*) to see whether or not the hit rate achieved by short simulation runs is typical of the hit rate achieved by runs that are too long to run on the simulator.

For each kind of filtering we plot the hit-rate achieved by four runs: "*Short (unbounded)*" uses the short workload, an unbounded filter size, and a 32MB heap that is sufficient to avoid GC on these short workloads. This provides a limit on the hit rate that could be achieved: in practice the hit rate will be lower because filter entries are lost (due to limited capacity) and cleared (due to GC).

"*Short (32MB)*" and "*Short (4MB)*" show the hit rate for the short workload with a fixed-size 512x4-entry filter. No GC occurs in the short 32MB runs, and so the difference between these and the unbounded runs shows the effect of conflict and capacity misses introduced by using a fixed-size filter. The difference between the 32MB and 4MB runs show the additional effect of reducing the young generation size (and consequently of clearing the filter upon each of the resulting GCs).

"*Long (4MB)*" shows the hit rate for the long workload with a 512x4-entry filter. A comparison between the long and short results shows whether or not the workload size affects the hit rate.

Figure 4(a) shows the results for using `dyfl_addr`. The hit rate achieved in realistic settings is much worse than the unbounded case. Consequently, the simple repetition check does not seem appropriate for this write-barrier.

Most strikingly, with Delaunay and XLisp, there is a loss going from an unbounded filter to a 512x4-entry filter, and a further loss from a 32MB heap to a 4MB heap. With Crafty, Delaunay, and XLisp there is a loss from the short workload to the long workload. This suggests that there is substantial repetition in the underlying workload (indicated by the hit rate for the unbounded runs), but the repetitions are sufficiently far apart in time that either (*i*) information from an earlier barrier has been lost from the filter before a later barrier (indicated by a higher hit rate in the unbounded case than in others), or (*ii*) a GC has occurred between the barriers, causing all entries to be flushed from the filter (indicated by a higher hit rate in the 32MB heap without GC, than the 4MB heap).

Figure 4(b) shows the results for the card-pair check. Aside from SatSolver, the results remain close to the unbounded runs: the card-pair check allows a 2048-entry filter to cover a large amount of the heap (reducing the difference between unbounded runs and the other runs), and the use of cards also allows filter state to exploit spatial locality to recover more quickly after a GC (reducing the difference between 32MB and 4MB runs).

We examined why SatSolver performs poorly with 4MB heaps. It frequently updates a set of fields in objects that are allocated near the start of execution. Once these are tenured in the old generation, each update potentially generates an old-to-young reference, and so the addresses cannot be added to the card-pair filter. The effect is more pronounced in the long workload because this phase of execution is proportionately larger.

Figure 4(c) shows the results when we combine two filters. The combination performs well: most access hit in the card-pair filter, and hot fields hit in the repetition filter once they have been logged. For all of the workloads we have studied, the "*Long (4MB)*" hit rate is close to the unbounded rate, and the hit rate on short, simulation runs is essentially the same as on long runs. We therefore use card-pair checks followed by repetition checks in subsequent GC results.

Performance. We now examine the performance of dynamic filtering for GC. We run all of these tests with sufficient heap space to avoid any GC work. Avoiding GC lets us focus only on the impact of write-barriers on the mutator.

Figure 5 shows the results (note the false origin). We compare three different configurations for each benchmark. The first configuration "*S/W*" uses ordinary software write-barriers. We normalize the results against this case. The second configuration "*Accel*" uses dynamic filtering, with 2048-entries (2048x1, 1024x2, 512x4), and with an unbounded filter size. The third configuration "*No GC*" is compiled without any support for GC, and consequently without including write-barriers in its implementation. This shows the maximum possible speed-up that could be achieved by accelerating the write-barriers. The only difference between the three configurations is the implementation of the barriers (if any): all three configurations use the same bump-pointer allocator, and the compiler was configured to ignore the size of any barriers when considering whether or not to inline methods.

The 2048x1 results are close to the "*No GC*" results for all of the benchmarks. This shows that dynamic filtering comes close to the ideal. The total possible impact is dependent on the number of heap updates in the program, and the benchmarks where dynamic filtering has the largest impact are those where there are most updates (Figure 2). There is little additional benefit from more associativity, or from a hypothetical unbounded filter size.

4.4 STM with Dynamic Filtering

Bartok-STM [2, 20] uses eager version management (that is, transactions make updates directly to the heap, and roll them back on conflict). It uses encounter-time object-based locking for updates, keeping a per-transaction log of objects that have been locked. It uses version-number-based validation for objects that are read by

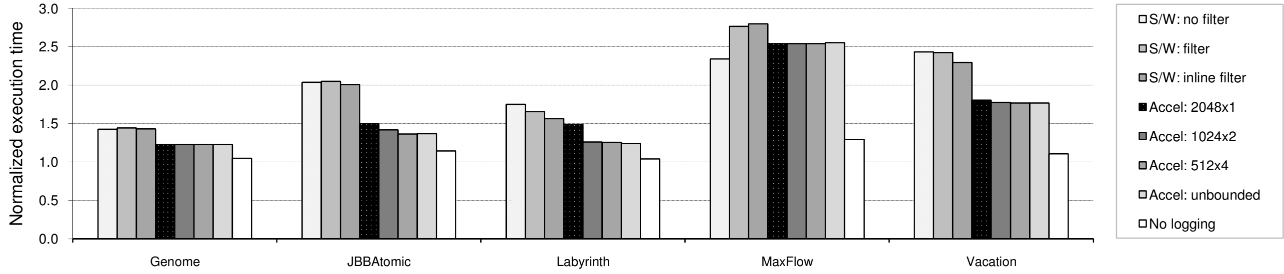


Figure 6. STM acceleration performance, normalized against the unsynchronized version of each benchmark. The extreme right-most bar bounds performance by treating each operation as a filter hit, causing no logging to be performed.

a transaction: the version number is logged before the transaction reads from the object, and then the version number is checked again when the transaction commits. An undo-log records the individual words that a transaction overwrites.

Transaction management operations are eligible for inlining in the usual way. Data logging operations are made using manually-written sections of the compiler's intermediate code (which make calls to non-inlined slow-path operations on log overflow or lock contention). There are three data-logging operations:

```
OpenForRead(Object x)
OpenForUpdate(Object x)
LogForUndo(Object x, int field_offset)
```

The `Open*` operations must be placed before the first read/update access to an object. A `LogForUndo` operation must be placed between `OpenForUpdate`, and the first write to a particular word. All three of these operations are only needed before the *first* access to a given object or location—subsequent repetitions can be removed. (Concretely, similar operations exist for access to statics, and indirect access to by-ref parameters. For brevity we do not discuss these further.)

Existing static analyses. The Bartok compiler uses simple intra-procedural data flow analyses to remove STM operations that are redundant [20]. It also identifies transactional reads that dominate writes to the same object, and obtains write access before the read (avoiding recording the same object in both logs). Logging operations on loop-invariant locations can be hoisted from loops (e.g., if each iteration reads from a different element of the same array, then the array can be logged outside the loop). The compiler identifies methods that always read/write the object that one of their parameters refers to and, for these methods, the `Open` operation is moved to the call site. Once at the call site, the `Open` operation may be redundant—e.g., because one method calls a series of simple `get`/`set` accessor methods on the same object.

Taken together, these existing analyses reduce the dynamic number of logging operations by 24% (Genome), 44% (JBBAtomic), 92% (Labyrinth), 36% (Maxflow), and 38% (Vacation). The analyses are enabled for all our results.

Dynamic filtering. The existing STM read/write-barrier implementations include a software fast-path which filters redundant operations [20]. This fast-path is implemented using per-thread tables, indexed by the bottom-few significant bits of the object's address or field's address. A stand-alone barrier would typically produce x86 native code as follows:

```
// Assume address to filter in eax
mov ebx <- fs:[filterBase]
mov ecx <- eax
and ecx <- 0x1ffc
cmp [ebx + ecx], eax
jnz slow_path
```

In this code, the load from `filterBase` accesses a location in the thread-local `fs` segment. Masking with `0x1ffc` converts the address in `eax` into an offset into a table of 2048 pointer-sized values. The comparison succeeds if the address `eax` was the last one to be logged at that offset. The slow-path stores `eax` into the table before doing any STM logging work. The table must be cleared on each transaction. To do this efficiently, the existing software (*i*) allocates a double-sized table, (*ii*) uses a sliding section of this, indexed by `filterBase`, (*iii*) increments `filterBase` by one entry on each transaction, and (*iv*) clears the double-size table on reaching the end of it. (Adjusting the base means that, with this simple direct-mapped scheme, entries written by one transaction will not be misused by a second transaction.)

To use hardware dynamic filtering, we replace this sequence with a `dyfl` instruction. We use tags to distinguish reads from writes, adding both filter entries on an `OpenForUpdate` miss. We perform dynamic filtering based on object header addresses for `OpenForRead` and `OpenForUpdate`. We use filtering based on field addresses for `LogForUndo`.

Performance. As with GC, we compare three kinds of STM implementation: ordinary software implementations, accelerated implementations using `dyfl`, and a bounding case with logging work removed. We look at single-threaded performance, letting us focus on the overheads introduced by the different barrier implementations. (The actual logging and conflict detection is identical for all implementations, so scaling is unchanged). Results are normalized against the performance of an unsynchronized version of the program—that is, one without any kind of concurrency control, neither locking nor TM. This baseline is not safe for parallel execution, but serves as a bound on the possible sequential performance (unlike a lock-based version, whose performance is dependent on the lock's implementation, as well as the underlying program).

Figure 6 shows the results for the benchmarks that use transactions. The first set of bars show software implementations. "*S/W: no filter*" shows direct use of the STM library, without the fast-path software filtering. "*S/W: filter*" shows the effect of adding the filtering, but in a separate function rather than inlined in the application. "*S/W: inline filter*" shows the default software configuration, with the barrier fast-paths inlined. The second set of bars (starting with the black bar) shows the performance of implementations using `dyfl`. These use a 2048-entry filter configured as 2048x1, 1024x2, and 512x4-entry sets. The third set of bars provide bounds on performance. "*Accel: unbounded*" uses a filter of unbounded size. Comparing this with the previous "*Accel*" results shows how much performance is lost by having a fixed-sized filter and by having limited associativity. "*No logging*" uses an unsound `dyfl` implementation in which every filter access is treated as a hit. This shows the residual effect of the transaction management operations (start/commit).

	Hit rate (%)	Dynamically polarized instructions never-log (%)	always-log (%)
Genome	59.8	7.6	9.4
	59.6	7.7	9.5
JBBAtomic	77.0	48.4	6.9
	73.2	21.9	9.0
Labyrinth	75.3	51.6	1.0
	75.3	22.3	0.1
MaxFlow	1.4	0.0	96.5
	1.2	0.0	96.8
Vacation	59.1	17.5	3.6
	58.8	16.8	3.8

Figure 7. Redundancies exploited by dynamic filtering. A *polarized* instruction is one which always hits or always misses.

Dynamic filtering substantially reduces the cost of STM in 4 of the 5 benchmarks. Looking at the cost above the 1.0 normalized baseline, the 512x4 results show a reduction of 47% in Genome, 64% in JBBAtomic, 55% in Labyrinth, and 40% in Vacation when compared against the inlined software barriers.

Dynamic filtering is less effective in MaxFlow. The equivalent performance reduction is 14%. MaxFlow comprises a vast number of short, simple transactions. The static analyses remove all the redundant operations, and so the inlined barriers with software filtering are *slower* than simply using the STM library directly. In the longer term, this kind of short-transaction workload could be handled by the support for limited-size hardware transactions which is emerging in industrial designs [13].

Comparing the unbounded results with the 512x4-entry filters shows that there is little additional benefit in using a larger filter size, or from having greater associativity.

Redundancies exploited by dynamic filtering. We tried to quantify whether dynamic filtering was finding "easy" forms of redundancy that might be identified by additional static analyses.

To do this, we instrumented each `dyfl` instruction to record the number of hits and misses that occur with an unbounded filter size. From this, we can identify instructions that always generate filter hits, or always generate filter misses. We call these "polarized" instructions. For a given run, polarized instructions which generate filter hits would have been safe to remove. Of course, this is not an exact indication of where `dyfl` instructions could be removed by static analyses—e.g., an instruction may be redundant in one run but not in another, or it may be that code motion can change a polarized `dyfl` instruction into a non-polarized one, or *vice-versa*. However, a high fraction of polarized instructions would intuitively suggest that further static analyses might be straightforward.

Figure 7 shows the results for the STM benchmarks. As in Figure 2, the first row for each benchmark shows the simulation workload, and the second row shows the longer workload. The *hit-rate* column provides the overall `dyfl` hit rate. The *never-log* column shows the (dynamic) percentage of instructions that never generate log entries. These results show that, for long workloads, the hit rate achieved by `dyfl` is not simply due to polarized instructions. The never-log percentage is higher for short workloads. This is because some `dyfl` instructions generate log entries very occasionally, and so they seem polarized on short runs.

Figure 7 also includes an *always-log* column, showing polarized instructions that always generate log entries. Future implementations could avoid using `dyfl` entirely for these instructions. This would be useful, in particular, with MaxFlow.

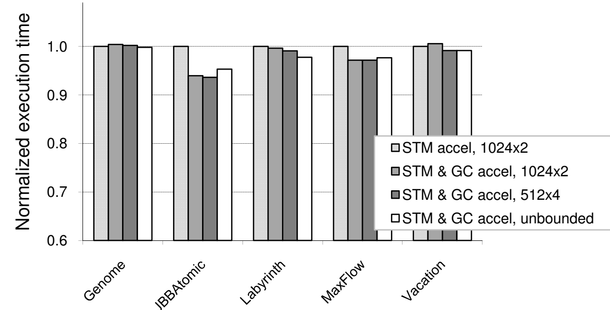


Figure 8. Combined performance, normalized against the accelerated STM performance.

4.5 Combined use

We now look at how dynamic filtering can be used for STM and GC at the same time. The two applications use different sets of tags, but they contend for space in the same set of filter entries without any policy to reserve space for one use or the other.

Figure 8 shows the results, normalized against the 1024x2-entry filter applied only to STM, adding GC acceleration with 1024x2 filtering, 512x4 filtering, and unbounded filtering. As before, the results with 2048-entry filters are close to the unbounded size.

The savings from GC acceleration in Figure 8 are proportionately less than those in Figure 5. However, this simply reflects the fact that the earlier results use the sequential unsynchronized versions of the programs, while the latter results use the STM versions. The absolute amount saved is the same in the two cases, but the saving is proportionately smaller in the latter case because of the additional time spent in the STM implementation.

4.6 Sensitivity

In Sections 4.3–4.5 we have used 2048-entry filters with varying degrees of associativity. The close match between these results, and those with an unbounded filter size, suggests that these workloads would obtain little benefit in providing a larger filter size.

We studied `JBBAtomic` and `MaxFlow` in more detail to see whether or not the results are sensitive to the use of 2048-entry filters, or the 512-byte card size.

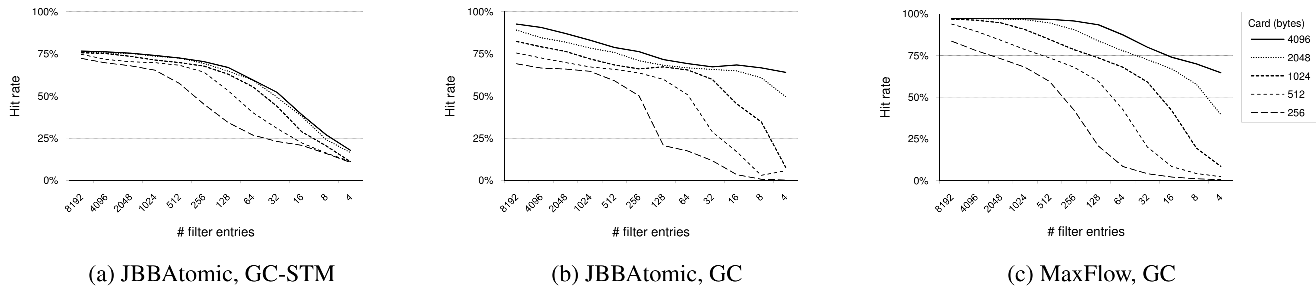
Figure 9 shows the results, using dynamic filtering for GC and STM (JBBAtomic), and just for GC (JBBAtomic, MaxFlow). For brevity, we omit STM results for MaxFlow because, as shown in Figure 7, the STM hit rate for MaxFlow is close to 0, irrespective of the filter configuration. We vary the filter size from 8192 entries down to 4 entries, and the card size from 4096-bytes down to 256-bytes. These results suggest that if we use a larger card size then the total number of filter entries could be reduced from 2048.

5. Related Work

In this section we discuss related work on hardware support for garbage collection (Section 5.1), transactional memory (Section 5.2), and for language-based security (Section 5.3). At a high level, there are two specific contributions of `dyfl` over existing approaches for filtering redundant barrier operations:

First, `dyfl` identifies barrier operations by *pairs* of addresses, whereas existing work identifies barriers by a single address. Handling pairs of addresses exposes many more potential applications—e.g., the garbage collection example from Section 4.3, and many of the language-based security examples we sketched in Section 3.

Second, `dyfl` generalizes existing filtering approaches by providing quantization of ranges of addresses. This is important for



(a) JBBAtomic, GC-STM (b) JBBAtomic, GC (c) MaxFlow, GC

Figure 9. Sensitivity of results to the filter size and card size.

many of the heap management examples from Section 3, because it allows groups of similar addresses to be handled by a single filter entry. Our results from Figure 4 show how filtering based on quantized pairs of addresses lets us achieve a significantly higher hit rate than a single-address repetition filter.

5.1 Garbage Collection

Hardware support for tagged values has often been used by garbage collected languages. Moon describes garbage collection in the Symbolics 3600 LISP implementation [30]. Tagged memory identifies locations containing references, and a hardware read barrier uses a table look-up to detect when the mutator loads a pointer to from-space. A hardware write-barrier maintains a log of young objects that have been made reachable from other objects. MASA [19] provides specific forms of "generation trap" and "transport trap" interrupts for generational GC and incremental GC. Meyer proposed architecture support for a particular kind of concurrent-GC read-barrier for an object-based RISC processor [28]. Ungar investigated support for a write barrier for generational GC as one feature of the SOAR processor [39]. This is based on tags that identify each object's generation.

Object-based processors have included support for GC. Williams and Wolczko's "Mushroom" architecture [41, 43] supports collection of the current cache contents (akin to a young-space collection in a generational system). Cache misses are managed by a software trap handler which can maintain a remembered set of objects that become reachable from outside the cache.

Wright *et al.* describe architecture support for garbage-collected object-oriented programming [44]. As with Mushroom, they use an object-addressed cache. A garbage collector may relocate an object by copying its physical representation without changing its object-id. As with Peng and Sohi's work [31], a zero-and-allocate instruction is provided to avoid needing to explicitly clear memory which will be overwritten by a constructor. A further instruction is provided to avoid writing known-to-be-garbage data back to memory. Wright *et al.*'s design supports in-cache garbage collection, in which a set of cores are paused and GC is performed across their local caches (e.g., a whole CMP might be paused together, but different CMPs in the same system would work independently). A *non-local bit* in an object header's cache line indicates that a reference to that object may have left a GC boundary.

Heil and Smith [21] exploit profiling hardware to log information about stores to the heap. Dieckmann and Hölzle investigated the use of active memory systems to support GC [17]. Chang *et al.* investigated architecture support for a bitmap-based allocator and mark-sweep garbage collector [12].

Click *et al.* describe techniques used by Azul System's parallel hardware to support "pauseless" garbage collection in an implementation of the Java Virtual Machine [15]. There are a number of mechanisms. First, the hardware supports a "GC-mode" access setting, in between the usual user-mode / kernel-mode distinction. GC-mode data is accessible to a collector, but inaccessible to mutator code. Second, variants of some common instructions (e.g., backwards branches, and function calls) can be flagged as GC-safepoints at which per-CPU safepoint interrupts are delivered. Third, a specialized form of hardware read-barrier is provided. The barrier is used after a normal load to validate the value that has been loaded from memory. It branches to a trap handler if the value loaded is on a GC-mode page, or if a "not-marked-through" bit in the value has the wrong sense.

Joao *et al.* describe a technique for hardware-supported reference-counting [24]. The ISA provides separate instructions for manipulating references, and for interacting with a storage allocator. The implementation tracks reference counts in the L1 and L2 caches, using a new hardware structure to coalesce updates to a reference count field, and detecting if the count reaches zero. This can be used in an elegant combination with a traditional software GC to provide fast reuse of memory within a GC cycle.

5.2 Transactional Memory

Saha *et al.* examined architectural support for software transactional memory [34]. They add a "mark" bit to blocks of memory and provide mechanisms for these to be set and tested. The mark-bits are non-persistent; for example, they may only be held for lines in the L1-cache. Additional user-accessible meta-data indicates when mark-bit values are lost. Our approach generalizes Saha *et al.*'s work by allowing filtering to be based on address-pairs, and allowing the addresses involved to be quantized.

Click argued that processors should provide a user-accessible bit which is set whenever any data access misses in the processor's L1-cache, and whenever any line is evicted from the L1-cache [14]. He showed how this can be used to support cache-resident snapshot operations, and that additional atomic operations can be supported if a store can be made conditional on the cache bit.

Baugh *et al.* used fine-grain protection to isolate transactional data in an implementation of TM with strong atomicity [6], and to separate hardware-managed and software-managed regions in a hybrid system. Baugh *et al.*'s approach does not aim to accelerate longer transactions executed in STM; it is a good complement to our work, since longer transactions tend to exhibit more repetition.

Yen *et al.* [46] and Sanyal *et al.* [35] describe hardware mechanisms to avoid logging thread-private data (e.g., stacks). That kind of filtering is complementary to the kind we study here. With Bartok-STM, stack accesses are typically made with different MSIL bytecodes which do not introduce logging in the first place.

5.3 Security and Isolation

Horowitz *et al.* investigated "informing loads" [22] which combine a memory access with a conditional branch dependent on cache miss. They examined different mechanisms for making the control

flow transfer—e.g., an explicit branch, squashing an instruction in a load-delay slot, or using an implicit lightweight trap to a pre-defined address. The latter approach reduced the cost on the (typically common) case of cache hits.

Witchel's "Mondriaan" memory protection mechanism support fine-grained control over memory protection settings [42] to support many of the software-security scenarios enabled by XFI [38].

Zhou *et al.*'s "iWatcher" supports user-supplied tracing functions that execute when an application accesses memory [47]. These can be used to identify memory leaks, buffer overflows and stack-corruption. These checks are heavier-weight operations than we aim to support with dynamic filtering, with transitions to the tracing functions made implicitly on accesses.

6. Conclusion

In this paper we have introduced dynamic filtering as an abstraction for accelerating read/write-barriers used by language runtime systems. The key idea is to provide a mechanism for testing whether or not a given runtime check has already been made—if the check has already been performed then the barrier can be elided. If the check has not yet been performed then the barrier executes as normal. We have shown that GC and STM workloads frequently exhibit sufficient spatial and temporal locality in their accesses that these checks can avoid significant numbers of barrier executions. This approach contrasts with previous work which have developed hardware replacements for specific forms of barrier, and which are consequently specialized to particular memory layouts or algorithms.

In future work we wish to evaluate the use of dynamic filtering more broadly. We sketched a number of such uses in Section 3, and would like to implement more of these. There are numerous policy-related design choices which we hope to revisit with reference to a wider range of applications of `dyfl`. First, when using `dyfl` in several ways at the same time, we may need to revisit the choice to use a small number of statically-allocated tags; some kind of dynamic arbitration may be necessary. Techniques such as those for managing limited numbers of signature registers may be relevant here [37]. Second, some form of control may be needed over the portion of the filter state which is devoted to any given tag—particularly if one tag is flooding the filter state without any benefit.

We would also like to evaluate the performance on different compilers, and machines based on different instruction sets. For instance, a system using a dynamic compiler might benefit from using `dyfl` more significantly than the ahead-of-time compiler that we have used. This is because dynamic compilation might have less time to devote to optimizing barrier placement in software, and so the amount of redundancy available dynamically might be greater. We would also like to examine the performance of practical implementations, e.g., using the RAMP BEE3 platform [16].

Acknowledgments

We would like to thank Martín Abadi, Mihai Budiu, Vladimir Gajinov, Steven Hand, Michael Isard, Peter Jonsson, Jean-Philippe Martin, Cristian Perfumo, Burton Smith, Karin Strauss and Ferad Zyulkyarov for their comments on earlier versions of this paper. We would also like to thank the anonymous reviewers for their feedback; many of the ideas for future work were suggested by the reviewers. This work is supported by the cooperation agreement between the Barcelona Supercomputing Center – National Supercomputer Facility and Microsoft Research, by the Ministry of Science and Technology of Spain and the European Union (FEDER funds) under contract TIN2007-60625, and by the European Network of Excellence on High-Performance Embedded Architecture and Compilation (HiPEAC).

References

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity. In *CCS '05: Proc. 12th ACM Conference on Computer and Communications Security*, pages 340–353, November 2005.

[2] Martín Abadi, Tim Harris, and Mojtaba Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. In *PPoPP '09: Proc. 14th ACM Symposium on Principles and Practice of Parallel Programming*, pages 185–196, February 2009.

[3] Periklis Akritidis, Cristian Cadar, Costin Raiciu, Manuel Costa, and Miguel Castro. Preventing memory error exploits with WIT. In *SP '08: Proc. 2008 IEEE Symposium on Security and Privacy*, pages 263–277, May 2008.

[4] Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark F. Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. The Jikes research virtual machine project: building an open-source research community. *IBM Syst. J.*, 44(2):399–417, 2005.

[5] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *OOPSLA '03: Proc. 18th ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*, pages 269–281, October 2003.

[6] Lee Baugh, Naveen Neelakantam, and Craig Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *ISCA '08: Proc. 35th International Symposium on Computer Architecture*, pages 115–126, June 2008.

[7] Stephen M. Blackburn and Antony L. Hosking. Barriers: friend or foe? In *ISMM '04: Proc. 4th International Symposium on Memory Management*, pages 143–151, October 2004.

[8] Stephen M. Blackburn and Kathryn S. McKinley. In or out?: Putting write barriers in their place. In *ISMM '02: Proc. 3rd International Symposium on Memory Management*, pages 175–184, June 2002.

[9] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[10] Mihai Budiu, Úlfar Erlingsson, and Martín Abadi. Architectural support for software-based protection. In *ASID '06: Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability*, pages 42–51, October 2006.

[11] Miguel Castro, Manuel Costa, and Tim Harris. Securing software by enforcing data-flow integrity. In *OSDI '06: Proc. 7th Symposium on Operating Systems Design and Implementation*, pages 147–160, November 2006.

[12] J. Morris Chang, Witawas Srisa-an, Chia-Tien Dan Lo, and Edward F. Gehringer. DMMX: dynamic memory management extensions. *Journal of Systems and Software*, 63(3):187–199, 2002.

[13] Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay. Rock: A high-performance Sparc CMT processor. *IEEE Micro*, 29(2):6–16, 2009.

[14] Cliff Click. IWannaBit! In *MSPC '08: Proc. 2008 ACM SIGPLAN workshop on Memory Systems Performance and Correctness*, pages 20–25, March 2008.

[15] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In *VEE '05: Proc. 1st International Conference on Virtual Execution Environments*, pages 46–56, June 2005.

[16] John D. Davis, Charles P. Thacker, and Chen Chang. BEE3: Revitalizing computer architecture research. Technical Report MSR-TR-2009-45, Microsoft Research, April 2009.

[17] Sylvia Dieckmann and Urs Hölzle. A case for using active memory to support garbage collection. In *Proc. 1st Workshop on Hardware Support for Objects and Microarchitectures in Java*, pages 1–5, October 1999.

[18] Damien Doligez and Xavier Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In

POPL '93: Proc. 20th ACM Symposium on Principles of Programming Languages, pages 113–123, January 1993.

[19] Robert H. Halstead, Jr. and Tetsuya Fujita. MASA: a multithreaded processor architecture for parallel symbolic computing. *SIGARCH Comput. Archit. News*, 16(2):443–451, 1988.

[20] Tim Harris, Mark Plesko, Avraham Shinnar, and David Tarditi. Optimizing memory transactions. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 14–25, June 2006.

[21] Timothy H. Heil and James E. Smith. Concurrent garbage collection using hardware-assisted profiling. In *ISMM '00: Proc. 2nd International Symposium on Memory Management*, pages 80–93, October 2000.

[22] Mark Horowitz, Margaret Martonosi, Todd C. Mowry, and Michael D. Smith. Informing memory operations: memory performance feedback mechanisms and their applications. *ACM Trans. Comput. Syst.*, 16(2):170–205, 1998.

[23] Galen C. Hunt and James R. Larus. Singularity: rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2):37–49, 2007.

[24] José A. Joao, Onur Mutlu, and Yale N Patt. Flexible reference-counting-based hardware acceleration for garbage collection. In *ISCA '09: Proc. 36th International Symposium on Computer Architecture*, pages 418–428, June 2009.

[25] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996.

[26] Haim Kermany and Erez Petrank. The compressor: concurrent, incremental, and parallel compaction. In *PLDI '06: Proc. 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 354–363, June 2006.

[27] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for Java. *ACM Trans. Program. Lang. Syst.*, 28(1):1–69, 2006.

[28] Matthias Meyer. A true hardware read barrier. In *ISMM '06: Proc. 5th International Symposium on Memory Management*, pages 3–16, June 2006.

[29] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proc. IEEE International Symposium on Workload Characterization*, pages 35–46, September 2008.

[30] David A. Moon. Garbage collection in a large LISP system. In *LFP '84: Proc. 1984 ACM Symposium on LISP and Functional Programming*, pages 235–246, August 1984.

[31] Chih-Jui Peng and Gurindar S. Sohi. Cache memory design considerations to support languages with dynamic heap allocation. Technical Report 860, Wisconsin CS Dept, 1989.

[32] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *PLDI '08: Proc. 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 33–44, June 2008.

[33] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 187–197, March 2006.

[34] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO '06: Proc. 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 185–196, December 2006.

[35] Sutirtha Sanyal, Sourav Roy, Adrián Cristal, Osman S. Unsal, and Mateo Valero. Dynamically filtering thread-local variables in lazy-lazy hardware transactional memory. In *HPCC '09: Proc. 11th IEEE International Conference on High Performance Computing and Communications*, June 2009.

[36] Michael L. Scott, Mike F. Spear, Luke Dalessandro, and Virendra J. Marathe. Delaunay triangulation with transactions and barriers. In *IISWC '07: Proc. IEEE International Symposium on Workload Characterization*, pages 107–113, September 2007.

[37] James Tuck, Wonsun Ahn, Luis Ceze, and Josep Torrellas. SoftSig: software-exposed hardware signatures for code analysis and optimization. In *ASPLOS '08: Proc. 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 145–156, March 2008.

[38] Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C. Necula. XFI: software guards for system address spaces. In *OSDI '06: Proc. 7th Symposium on Operating Systems Design and Implementation*, pages 75–88, November 2006.

[39] David Michael Ungar. *The Design and Evaluation of A High Performance Smalltalk System*. PhD thesis, EECS Department, University of California, Berkeley, Feb 1986.

[40] Martin T. Vechev and David F. Bacon. Write barrier elision for concurrent garbage collectors. In *ISMM '04: Proc. 4th International Symposium on Memory Management*, pages 13–24, October 2004.

[41] Ifor W. Williams and Mario I. Wolczko. An object-based memory architecture. In *POS '90: Proc. 4th Intl. Workshop on Persistent Object Systems*, pages 114–130. Morgan Kaufmann, September 1990.

[42] Emmett Jethro Witchel. *Mondriaan memory protection*. PhD thesis, Massachusetts Institute of Technology, January 2004.

[43] Mario Wolczko and Ifor Williams. Multi-level garbage collection in a high-performance persistent object system. In *POS '92: Proc. 5th Intl. Workshop on Persistent Object Systems*, pages 396–418, September 1992.

[44] Greg Wright, Matthew L. Seidl, and Mario Wolczko. An object-aware memory architecture. *Sci. Comput. Program.*, 62(2):145–163, 2006.

[45] Luke Yen, Jayaram Bobba, Michael R. Marty, Kevin E. Moore, Haris Volos, Mark D. Hill, Michael M. Swift, and David A. Wood. LogTM-SE: Decoupling hardware transactional memory from caches. In *HPCA '07: Proc. 13th International Symposium on High Performance Computer Architecture*, February 2007.

[46] Luke Yen, Stark C. Draper, , and Mark D. Hill. Notary: Hardware techniques to enhance signatures. In *MICRO '08: Proc. 41st International Symposium on Microarchitecture*, pages 234–245, November 2008.

[47] Yuanyuan Zhou, Pin Zhou, Feng Qin, Wei Liu, and Josep Torrellas. Efficient and flexible architectural support for dynamic monitoring. *ACM Trans. Archit. Code Optim.*, 2(1):3–33, 2005.