The Future(s) of Shared Data Structures

Alex Kogan
Oracle Labs
Burlington, MA, USA
alex.kogan@oracle.com

Maurice Herlihy[*]
Brown University and Oracle Labs
Providence, RI, USA
mph@cs.brown.edu

ABSTRACT

This paper considers how to use futures, a well-known mechanism to manage parallel computations, to improve the performance of long-lived, mutable shared data structures in large-scale multicore systems. We show that futures can enable type-specific optimizations such as combining and elimination, improve cache locality and reduce contention. To exploit these benefits in an effective way, however, it is important to define clear notions of correctness. We propose new extensions to linearizability appropriate for method calls that return futures as results. To illustrate the utility and trade-offs of these extensions, we describe implementations of three common data structures: stacks, queues, and linked lists, designed to exploit futures. Our experimental results show that optimizations enabled by futures lead to substantial performance improvements, in some cases up to two orders of magnitude, compared to well-known lock-free alternatives.

Categories and Subject Descriptors

E.1 [**Data**]: Data Structures—*lists, stacks, and queues*; D.1.3 [**Software**]: Programming Techniques—*concurrent programming*

General Terms

Algorithms, Design, Theory

Keywords

Concurrent data structures; futures; elimination; combining; linearizability

1. INTRODUCTION

Futures [5] are widely considered an attractive way to manage parallel computations: they are simple to use, and they lend themselves well to efficient scheduling mechanisms such as work-stealing. In this paper, we consider the use of futures for different purposes: improving the performance of long-lived concurrent data structures in large-scale multicore systems.

We are interested in the design and implementation of long-lived objects shared by many threads. Calling an operation of such an object might take a long time, perhaps because there is contention, or because the object resides on a remote node in a non-uniform memory access (NUMA) architecture. One way to alleviate these problems is to have the object's operations return futures as return values. By using futures, threads can accumulate multiple pending operations, that is, operations that were invoked, but not yet applied to the object. These pending operations can be evaluated in a flexible way, perhaps exploiting type-specific optimizations such as combining and elimination intended to reduce contention, improve cache locality and reduce communication costs.

Our use of futures in this paper departs in important ways from the original proposal, where futures were benign annotations that did not affect a program's meaning, and future computations had no side-effects [5]. By contrast, our use of futures centers on operations that exist for their side-effects on shared, long-lived state, operations that can interact and interfere with one another.

As a result, it is essential to define clear and useful notions of correctness for objects whose operations return futures. A major observation of this paper is that no single notion of correctness is likely to cover all reasonable cases. Instead, we propose three new extensions to linearizability [11], called *strong*, *medium*, and *weak futures linearizability*. Each notion makes a different trade-off between determinism, which reduces the number of possible interleavings, and the flexibility to apply various optimizations, which yields more efficient programs. Like standard linearizability, however, these notions require that any concurrent execution be equivalent, in some sense, to a sequential computation.

To demonstrate the utility of these notions, we describe strong, medium, and weak futures linearizable implementations of three common data structures: stacks, queues, and linked lists, and compare their performance on a multicore system to well-known lock-free algorithms. These experiments illustrate in a quantitative way the trade-offs between performance and determinism. In particular, we show that implementations satisfying the weak and medium correctness conditions typically significantly outperform lock-free alternatives, in some cases by up to two orders of magnitude.

[*]Supported by NSF 1301924.

```
Future<T> f = submit(task); // start task
 ...                        // do something else
T x = f.eval ();            // pick up task's result
```

Figure 1: Use of futures

Even though the versions satisfying the strong correctness condition typically perform worse than other future-based versions, we show cases where they achieve better performance than the lock-free alternatives.

The rest of the paper is organized as follows. We discuss the benefits of using futures with long-lived concurrent data structures in Section 2. In Section 3 we informally define three extensions to linearizability that support operations returning futures. We exploit these extensions by constructing three common data structures in Section 4. These constructions are evaluated and compared to lock-free alternatives in Section 5. The formal model and proofs appear in Section 6. The related work is surveyed in Section 7. Finally, we conclude the paper with discussion in Section 8.

2. FUTURES AND LONG-LIVED OBJECTS

A *future* [5] is a data object that represents a *promise* to deliver the result of an asynchronous computation when it is ready. Figure 1 shows a simple code fragment that uses a future. In Line 1, a thread creates a future to hold the result of an asynchronous computation. In Line 2, the thread does something unrelated, and in Line 3, it calls the future's eval() method to retrieve the task's result, blocking if necessary until that result is ready. (Calling a future's eval() method is called *evaluating* or *touching* that future.)

While futures were originally proposed as a way of managing short-lived functional computations, here we focus on long-lived mutable objects, like maps, lists, queues, or stacks, shared by multiple threads in a multicore architecture. These objects return future values in response to method calls. These method calls typically have side-effects (such as binding a key to a value), and the results may return information about the object state (such as the result of a map look-up) or confirmation that a side-effect has taken place (such as confirming that map has finished binding a key to a value).

The principal contribution of this paper is the observation that futures, properly redefined, can provide substantial benefits to long-lived concurrent data structures. These benefits include:

- *Combining*: multiple operations may be combined into a single operation on the shared object, reducing computation and communication latency.
- *Elimination*: some operations may "cancel" each other out, eliminating the need to access the shared object at all.
- *Delegation*: one thread might be able to apply pending operations on behalf of others. This thread may be selected dynamically (as is done here) or it may be designated statically.
- *Contention reduction and cache efficiency*: combining and elimination may cause the shared data structure to be accessed less frequently, alleviating synchronization bottlenecks and incurring less cache coherence traffic.

```
Future<Nil> fx = queue.enq(x);
Future<Nil> fy = queue.enq(y);
Future<T> fz = queue.deq();
fx.eval ();              // force enq(x) to happen?
fy.eval ();              // force enq(x) to happen?
T z = fz.eval ();        // value at the head of queue?
```

Figure 2: A FIFO queue example

- *Scheduling*: the implementation is free to choose whether to apply pending method calls eagerly or lazily.

However, to apply these optimizations correctly, we need a careful definition what it means when a method call to a mutable object returns a future, and how a result retrieved from the evaluated future is related to the state of that mutable object.

3. FUTURES AND LINEARIZABILITY

A future's value is calculated sometime between when that future is created and when it is evaluated. As originally conceived, future computations were short-lived tasks without side-effects, so it was impossible to observe when these computations actually happened. Here, by contrast, we are interested in futures returned by methods that do have side-effects, such as adding or removing an item into or from a container. It is important to define when a future-returning method can "take effect", and, indirectly, how such effects might be interleaved.

In the absence of futures, it is common to define the behavior of concurrent method calls through properties such as linearizability [11]. Each object is assumed to provide a sequential specification, which describes its behavior in sequential executions where method calls do not overlap. To describe an object's concurrent behavior, each method call is split into two instantaneous events: the *invocation*, when control leaves the caller, and the *response*, when control (and perhaps a result) returns to the caller. Informally, an execution is *linearizable* if each method call appears to take effect instantaneously between its invocation and its response events.

To make effective use of futures, and to determine which optimizations are permitted, we will need to define extensions to linearizability. For ease of presentation, we define these extensions informally here. The formal definitions are postponed to Section 6.

There are several natural ways to extend linearizability to futures, and each has its own advantages and disadvantages. We will illustrate these choices using the code fragment in Figure 2, where a FIFO queue, initially empty, provides enq() and deq() methods that return futures. The thread first calls enq(x), returning future fx, then enq(y), returning future fy, and finally deq(), returning future fz. It then evaluates first fx, then fy, and finally fz. What can we assume about the value returned from fz? There are several reasonable choices.

Strong Futures Linearizability (Strong-FL).

One natural correctness condition is to require that each method call takes effect at some instant between its call (creating the future), and its response (returning the future), just as if it were a regular method call. In Figure 2, enq(x) precedes enq(y), which precedes deq(), so (assuming no other thread is accessing the queue) evaluating fz must yield x.

This property is easy to reason about, since it is essentially the same as linearizability, treating futures as benign hints for optimizations. Implementations are free to delay the actual computation until the future is evaluated, as long as no thread can tell the difference. As we show in Section 5, this separation of how operations are ordered from when they are evaluated can be very beneficial in some cases. For many applications, however, strong futures linearizability rules out certain attractive optimizations.

Weak Futures Linearizability (Weak-FL).

Another natural correctness condition is to require that each method call takes effect at some instant between its call (creating the future), and the return from that future's evaluation. In Figure 2, the calls to enq(x), enq(y), and deq() are free to take effect in any order, so evaluating fz could yield either Nil (if deq() takes effect first), x (enq(x) first), or y (enq(y) first). This choice gives the run-time system considerable freedom for optimization.

For some applications, however, this condition may be too permissive. A thread that wants to enqueue x before y would have to evaluate fx *before* calling enq(y), making optimizations such as combining impossible. Furthermore, if fx is not evaluated before calling enq(y), this condition may seem counter-intuitive, as (probably, a different) thread may see the effect of the second enqueue before the effect of the first one.

Medium Futures Linearizability (Medium-FL).

Here is an intermediate proposal that balances the programmer's guarantees with the flexibility for optimization. We separate two issues: (1) when a future's computation takes effect, and (2) how a future's computation is ordered with respect to other futures' computations. For question (1), we require each method call to take effect at some instant between that method's future's creation and its evaluation, just as in weak futures linearizability. For question (2), we require two future method calls issued by the *same thread* to the *same object* to take effect in the same order as their future creation operations. In Figure 2, enq(x) precedes enq(y), which precedes deq(), so evaluating fz must yield x. Notice that medium future linearizability does *not* constrain the perceived order of operations issued to distinct objects.

Informally, a correctness property is *non-blocking* if by itself it never forces a thread T with a pending operation to wait for another pending operation to complete before T can evaluate its own operation's future. In Section 6, we show that strong, medium, and weak futures linearizability are all non-blocking. Examples of correctness notions that are *not* non-blocking include *serializability* [1].

A correctness property is *compositional* if any combination of two or more objects, each individually satisfying that property, collectively also satisfies that same property. In Section 6, we show that strong, medium, and weak futures linearizability are all compositional.

Compositionality should not be taken for granted. For example, consider the following natural generalization of medium futures linearizability, which we call *futures sequential consistency*: (1) as before, each method call takes effect at some instant between that method's future's creation and its evaluation, but (2) two future method calls issued by the same thread *to any object* (not just an individual object) take effect in the same order as their future creation operations.

```
 1  Future<Nil> f1 = p.enq(x);
 2                                      Future<Nil> f2 = q.enq(y);
 3  Future<Nil> f3 = q.enq(x);
 4                                      Future<Nil> f4 = p.enq(y);
 5  f1.eval();                          f2.eval();
 6  f3.eval();                          f4.eval();
 7
 8  Future<T> fp = p.deq();
 9  fp.eval(); // returns y
10                                      Future<T> fq = q.deq();
11                                      fq.eval(); // returns x
```

Figure 3: Why futures sequential consistency is not compositional

Futures sequential consistency, like the classical notion of sequential consistency [12], is not compositional. Figure 3 shows a simple counterexample. Each column represents a thread, A and B, and the lines represent the interleaving. There are two FIFO queues, p and q. In Lines 1–4, A enqueues x on p, B enqueues y on q, A enqueues x on q, and B enqueues y on p. All four resulting futures are then evaluated (Lines 5–6). Futures sequential consistency requires that each thread's method calls take effect in the order their futures were created: Line 1 takes effect before Line 3, and Line 2 before Line 4. Since y is the first item dequeued from p, Line 4 takes effect before Line 1, and similarly, Line 3 takes effect before Line 2, resulting in a cyclic order. It is easy to check, however, that this interleaving is medium futures linearizable, because a thread's enqueue calls to different objects are unordered.

4. EXPLOITING FUTURES

This section describes how to use futures to encapsulate type-specific optimizations for three basic data types: queues, stacks, and linked lists (or more precisely, linked list-based sets). For brevity and simplicity, we treat all return values as futures. (Non-future return values can be treated as futures that are evaluated immediately.) We first describe high-level designs common to all three data types. In the following subsections, we provide additional details specific to each considered data type.

Threads share an instance of the data structure (queue, stack, or list, respectively). In weak and medium-FL implementations, this shared instance is similar to a lock-free version of the data structure[1], except that it supports single-operation insertion or removal of multiple nodes. For instance, pushing multiple nodes (organized in a list) into a stack is a straightforward extension of the single-node code, where the last node of the list is connected to the first node in the stack and the head of stack is updated with a single compare-and-swap (CAS) instruction to point the first node in the list. The strong-FL implementation employs a

[1]We use standard lock-free data structures from the literature: Michael and Scott's lock-free queue [14], Harris's linked list [6], and Treiber's stack [18].

sequential instance of the data structure, for reasons discussed below.

In our prototype implementations, futures are realized as objects with `opCode`, `value`, `result` and `resultReady` fields. The first two fields store the type of the corresponding operation (e.g., `enq()` or `deq()`) and an optional parameter. The evaluation of a future is completed by writing the corresponding result into the `result` field and setting the `resultReady` flag. An operation is *pending* if its invocation has occurred, but the operation itself has not yet been applied to its object. A thread may check quickly if an operation is still pending by reading the `resultReady` flag of that operation's corresponding future.

For the weak and medium-FL implementations, each thread uses thread-local list(s) to store and evaluate its own pending operations. Each such list presents an opportunity to apply type-specific optimizations (such as combining or canceling compatible operations). The exact details on how these local lists are maintained and used are given in the following subsections.

For the strong-FL implementations, each operation appears to take effect before its future is returned. To achieve that, all threads share a single queue of pending operations, which allows adding new operations (more precisely, future objects) in a lock-free manner. This queue is based on the lock-free queue by Michael and Scott [14]. When a thread invokes an operation, it allocates and enqueues a future object describing that operation, and returns the future to the caller. The evaluation of pending operations is protected by a lock. When a thread T evaluates a future F, it tries to acquire the lock. If it succeeds, and if F is still pending, T records the current last pending operation that is in the queue. (If T fails to acquire the lock, it waits until the lock becomes available again, checking periodically that F is still pending.) Then T evaluates all operations in the queue, including F, starting from the head (or, more precisely, from the operation next to the head, as the first operation is a dummy [14]) and up to the last recorded operation. Note that while T is holding the lock, other threads can keep adding new operations to the queue, but those operations might not be evaluated by T. However, the number of pending operations evaluated by T (and thus the time T possesses the lock) is limited. While evaluating pending operations, T is free to apply type-specific optimizations such as combining and eliminating multiple operations. Moreover, once T has exclusive access to the shared data structure, the (possibly combined) operations can be applied directly to the shared data structure instance, without any synchronization. When T is done, it updates the head of the queue to point the recorded last operation, effectively removing evaluated operations from the queue, and releases the lock.

The particular implementations described in this paper were chosen for simplicity. There are many other ways one could implement these objects while satisfying the correctness conditions presented in Section 3. For example, we note that our strong-FL implementations may have limited scalability, because the shared queue of pending operations or the lock protecting evaluations might become bottlenecks under high contention. Such limitations may matter more for objects whose operations are lightweight, such as queues and stacks. In particular, one might achieve better performance by applying the elimination optimization in the strong-FL stack before resorting to the shared queue of pending operations. In this paper, we opted to exploit futures by presenting an approach, which is not necessarily optimal, yet general enough to be applied for different data structures with minimal efforts. As we will see in Section 5, by enabling a clean separation between how operations are *ordered* from how they are *evaluated*, this approach is most beneficial for objects with heavyweight operations, such as long linked lists.

4.1 Stack Algorithm

Since the weak-FL condition permits each operation to take effect at any instant between its invocation and its future's evaluation, we can reorder pending `push()` and `pop()` operations, increasing the potential for the elimination optimization. As a result, we will never have both pending `push()` and `pop()` operations for the same thread at any given time. Thus, each thread maintains only one local list to store pending operations, each having the same type (that is, all operations in the local list are either `push()` or `pop()`).

When a thread invokes a `push()`, it checks whether its local list contains pending `pop()` operations. If so, the `push()` provides a result to one of the pending `pop()` operations. (That is, the `value` of the future for the `push()` operation is copied into the `result` field of the future for the `pop()` operation, and `resultReady` flag is set for both futures). Otherwise, the new `push()` is added to the list of pending `push()` operations. The code for `pop()` operations is symmetric.

When a thread evaluates a future, it actually evaluates all the futures stored in that thread's local list, in order to combine multiple operations and reduce traffic to the shared stack instance. Specifically, for `pop()` operations, as many items as there are pending `pop()` invocations are popped from the shared stack using a single CAS instruction. If the stack does not have enough items, all items are removed and excess `pop()` operations are paired with a special "empty stack" value. Symmetrically, the items associated with all pending `push()` operations are pushed onto the stack with a single CAS instruction.

The medium-FL stack cannot cancel complementary operations quite as aggressively since it has to respect the ordering of thread's operations. As a result, the local list may contain pending operations of distinct types. The medium-FL property implies that a `push()` operation cannot be combined with a prior pending `pop()` operation, but a `pop()` operation can be combined with the most recent prior `push()`. Multiple pending operations of the same thread and of the same type are combined before modifying the shared stack, allowing the thread to add or remove multiple items with a single CAS.

The strong-FL implementation is straightforward. When a thread attempts to evaluate a future, it tries to acquire the lock associated with the shared queue of pending operations. If it succeeds, it traverses that queue starting from its head. When possible, it eliminates pending `pop()` operations with preceding pending `push()` operations. At the end, it applies the remaining `pop()` and `push()` operations to the shared stack and releases the lock.

4.2 Queue Algorithm

Queue semantics does not allow operations to be eliminated as easily as stacks, yet it does allow for efficient combining. For the weak-FL queue, each thread keeps two thread-local lists of pending operations: one for pending enq() operations, and another for pending deq() operations. When a future returned by a method call is evaluated, so are all futures of the same operation type. This is again to combine a number of operations and reduce the number of accesses to the shared queue instance. The latter is achieved by inserting (removing) multiple nodes into (from) the shared queue, using just two (one, respectively) CAS operations.

For the medium-FL queue, each thread maintains one local list that contains all its pending operations in the order they were invoked. A thread evaluates a future F by traversing its local list, starting from the head (the oldest pending operation). It removes a sequence of pending operations of the same type (either enq() or deq()), combines operations in the sequence and applies those combined operations to the shared queue. This process is repeated until F is evaluated. Here, again, multiple nodes are inserted or removed to/from the shared queue with one or two CAS operations.

Similarly to stacks, the strong-FL implementation of queues is straightforward. A thread that acquires the lock, runs on the queue of pending operations and applies them to the shared queue.

4.3 Linked List Algorithm

The weak-FL linked list implementation keeps each thread's local list of pending operations sorted by key. A thread evaluates a future by traversing the shared list (which also maintains nodes in sorted order) and applying the pending operations from its local list. Multiple pending operations with the same key are combined, so that at most one modification per key is done to the global list. This implementation traverses the shared list just once to apply all pending operations.

Due to ordering restrictions on thread's operations, the medium-FL linked list implementation keeps each thread's local list of pending operations sorted in temporal order, not by key. A thread evaluates a future F by traversing the local list, starting from the oldest operation, and applying pending operations subject to the following optimization. In the shared lock-free list [6], all operations on the list employ an auxiliary search() function that accepts a key and traverses the list, looking for the last node holding a value less than or equal to the given key. When done applying a pending operation op(), if the next pending operation has a key larger than or equal to the key of op(), the thread resumes searching from the position in the list where op() was applied. Otherwise, and if F is not yet evaluated, it resumes searching from the head of the list.

Finally, in the strong-FL linked list implementation, a thread that evaluates a future acquires the shared lock. Next, it runs through the shared queue of pending operations, and sorts the operations by key. The sort is stable, meaning that the temporal (linearization) order of operations with the same key is preserved. Then it traverses the shared list just once, to apply the pending operations. Finally, it releases the lock.

5. EVALUATION

We evaluated the implementations described in Section 4 and compared them to well-known lock-free alternatives, namely Michael and Scott's lock-free queue [14], Harris's linked list [6], and Treiber's stack [18]. The implementations were done in C++. To avoid effects of memory management on our results, the memory required by any implementation was pre-allocated. We used a simple benchmark in which each thread performs a preset number (100K) of operations on a data structure, initialized as described below. No external work was performed by threads between operations on a shared data structure. The operations were chosen randomly from distributions described below.

A future-based implementation must choose how many pending operations to permit, a quantity we refer to as *slack*. We experimented with different choices of slack: after every X (=slack) operations returning futures (where X is 1, 10, 20 or 100), the thread evaluates all those futures before proceeding with the next X operations. For $X = 1$, each future is evaluated immediately, allowing a direct comparison between the overheads of the future-based versus lock-free implementations.

Our experiments were run on a 1-socket, 8-core SPARC T4 system featuring 64 hardware thread contexts and powered by Solaris 11 operating system. The sources were compiled by g++ 4.7.1 compiler with -O4 optimization level. We varied the number of threads between 1 and 64, and measured the time for all threads to complete their operations. The results shown are the mean of 10 runs performed with exactly the same parameters. The variance of the majority of the results is negligible.

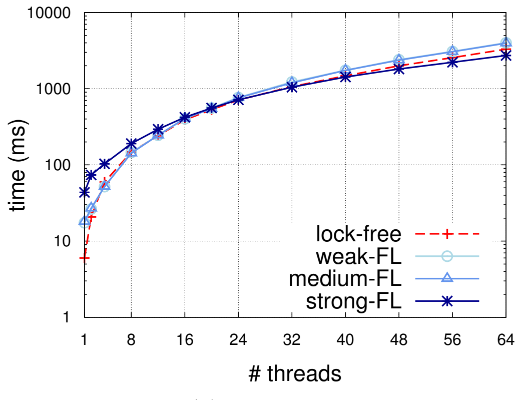
5.1 Stacks

Figure 4 shows the stack benchmark performance with different values for the slack (in log scale for y-axis). The stack is initially empty, and each thread executes 100K operations, choosing between push() and pop() with equal probability. As Figure 4(a) shows, with slack=1 and a single thread, futures-based stacks perform worse than the lock-free stack due to the additional overhead required for managing local lists of pending operations (for the weak-FL and medium-FL versions) or managing the shared queue and lock (for the strong-FL version). When the number of threads increases, this overhead is mitigated by the contention created by threads accessing the stack. Thus, in multi-threaded experiments with slack=1, futures-based stacks perform competitively with the lock-free implementation.

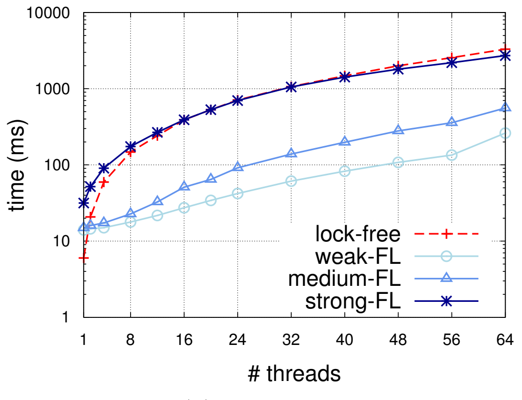
As the slack increases, the weak-FL and medium-FL stacks significantly outperform the lock-free stack, by up to two orders of magnitude. It is interesting to note that the gap between those two future-based stacks shrinks with the increase in slack. This is because the medium-FL stack manages to eliminate more operations, thus accessing the shared stack less frequently. Interestingly, the strong-FL stack performs competitively with the lock-free counterpart, and even slightly outperforms it on high thread counts. We believe that this is because the contention on the head of the stack in the lock-free stack is more severe than on the access to the shared queue of pending operations and on the lock protecting this queue in the strong-FL stack. We have seen the effect of one bottleneck relieving the contention on another bottleneck (and thus improving the overall performance) in other contexts as well [2].

5.2 Queues

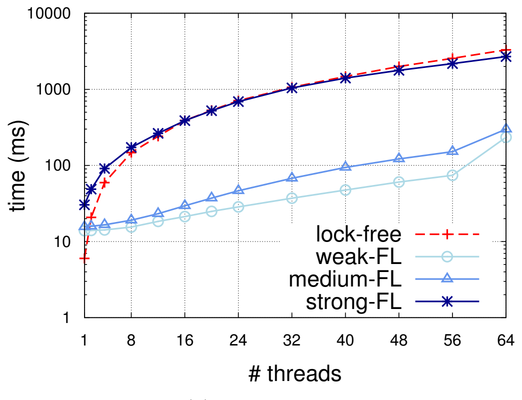
Figure 5 shows the queue benchmark performance for different levels of the slack. The queue is initially empty, and each thread executes 100K operations, choosing between enq() and deq() with equal probability. As with stacks, both



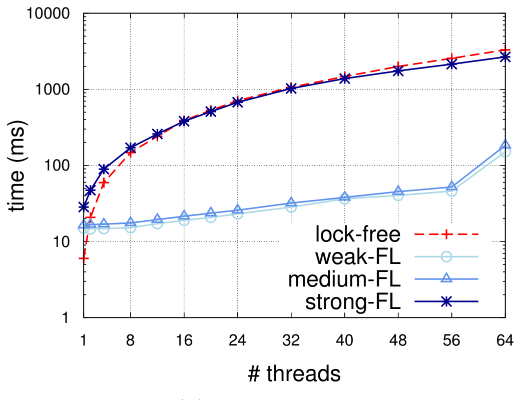
(a) slack = 1



(b) slack = 10



(c) slack = 20



(d) slack = 100

Figure 4: Stack benchmark performance

weak-FL and medium-FL queues perform similarly to the lock-free queue when the slack is just 1. In contrast, the strong-FL queue performs worse than the lock-free queue. The gap remains the same even when increasing the slack. The reasons are twofold. First, while in the strong-FL queue, all threads insert new pending operations at the tail of the queue of pending operations, the shared queue in the lock-free version is accessed from both ends, resulting in less contention. Second, queue semantics dictates that queue operations cannot be eliminated as aggressively as stack operations. Thus, the synchronization overhead of evaluating pending operations is not compensated by elimination.

The remaining queue performance phenomena shown in Figure 5 are also quite different from the stack results. First, the performance of the medium-FL queue relative to the lock-free queue stays the same for levels of slack larger than 1. This is because, in the medium-FL queue, future evaluation combines sequences of operations of the same type (cf. Section 4.2). Increasing the bound on the number of pending operations does not significantly increase the chance for larger sequences.

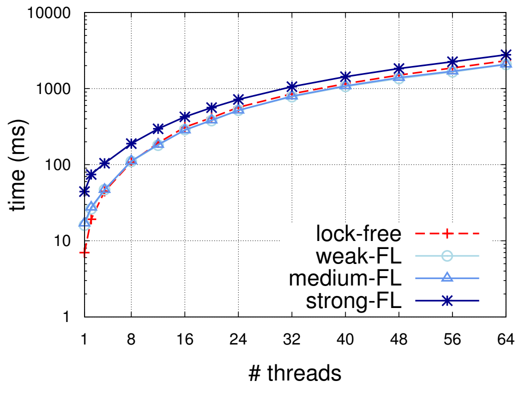
The weak-FL queue, however, does benefit from increasing the slack, as it can combine operations more aggressively. As the number of threads increases, however, this version's running time spikes sharply. It is interesting to note that the spike occurs at increasing thread counts as the slack increases. This happens because, even with aggressive combining, this version encounters contention on the shared queue with the increased number of threads. We validated this claim by measuring the average number of CAS operations issued by the weak-FL version per one high-level operation on the shared queue. We found out that there was strong correlation between the running time of the weak-FL version and the average number of CAS operations.

5.3 Linked Lists

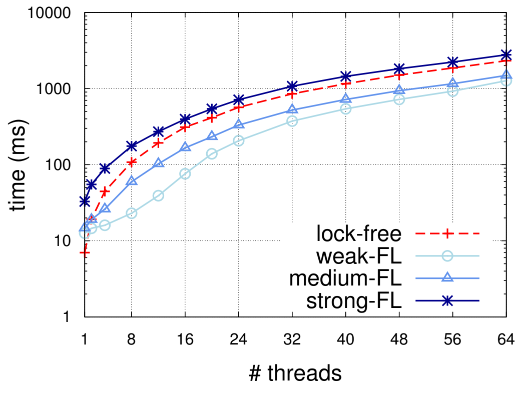
We evaluated the list benchmark performance in the following setting. The list is initialized with a number of randomly chosen keys. The number of keys is equal to half of the key range, which is set to 10K. Each thread executes 100K operations: 20% insert(), 20% remove(), and 60% contains(), chosen randomly. Each operation returns a Boolean flag indicating whether the list was changed or the required key was found. The results for different slack levels are shown in Figure 6.

Similarly to stacks and queues, the performance of the weak-FL list improves relatively to that of the lock-free list as the slack increases. This is because we combine more operations and apply them all by traversing the shared list just once. The medium-FL list achieves better performance than the lock-free list for slack larger than 1, but the relative gap stays constant. This is because, just as for queues, increasing the slack does not significantly increase the chance to apply more operations in one list traversal.

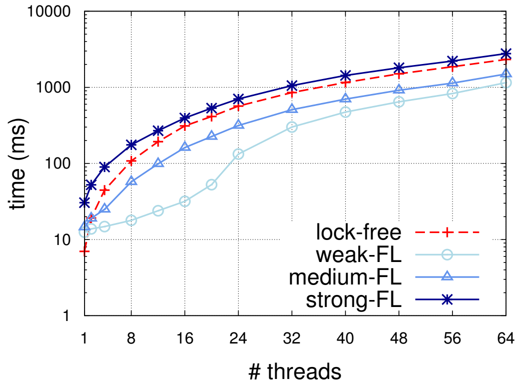
The strong-FL list shows some interesting behavior. As in the previous case, its performance is significantly worse rela-



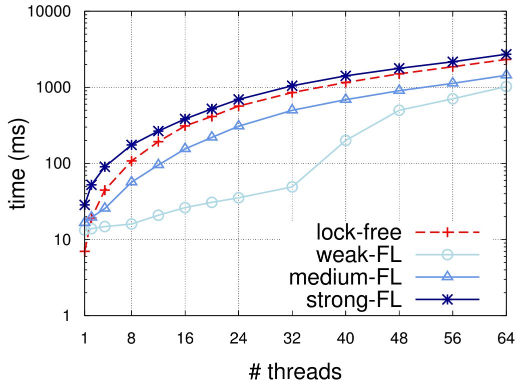
(a) slack = 1



(b) slack = 10



(c) slack = 20



(d) slack = 100

Figure 5: Queue benchmark performance

tive to all other versions when the slack is 1. However, once the slack increases, it beats the lock-free alternative (and the medium-FL list). This is because it enables one thread to effectively combine multiple pending operations produced by other threads, while those threads can continue to work in parallel on producing new operations. Given that the strong-FL property implies the medium-FL property, this example shows that, at least in our prototype implementations, the stronger correctness condition does not necessarily imply worse performance.

6. FORMAL MODEL

The formal definitions of the three kinds of futures linearizability are straightforward generalizations of the standard linearizability definition [10]. For simplicity and brevity, we assume that each future is evaluated at most once, and only by the thread that created it.

6.1 Model

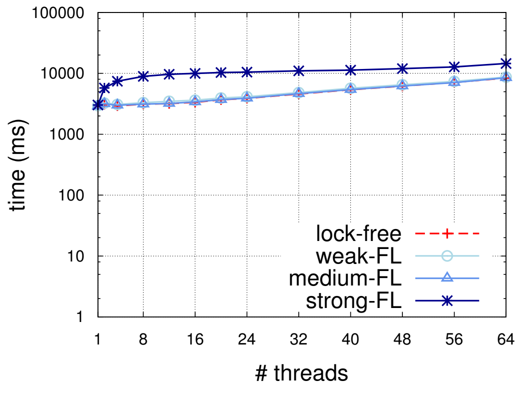
We first describe the standard linearizability model, and then discuss extensions to encompass the different degrees of futures linearizability.

Threads are *sequential*: they perform one method call at a time. (Later, we will relax this condition for method calls that return futures, allowing method calls issued by a single thread to overlap in constrained ways.) A method call is split into two events: an *invocation* event and a later *response* event. An execution of a concurrent system is modeled by a *history*, a finite sequence of method invocation and response events. A *subhistory* of a history H is a subsequence of the events of H. We write a method invocation as $\langle x\ m\ A\rangle$, where x is an object, m is a method (and arguments) and A is a thread. We write a method response as $\langle x\ t\ A\rangle$ where t is a value (possibly *void*) or an exception. Sometimes we refer to an event labeled with thread A as a *step* of A.

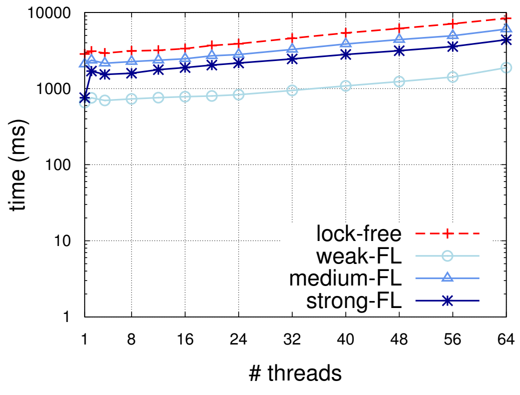
A response *matches* an invocation if they have the same object and thread. A *method call* in a history H is a pair consisting of an invocation and the next matching response. For a method call m, its delimiting events are denoted $inv(m)$ and $res(m)$. An invocation is *pending* in H if no matching response follows the invocation. An *extension* of H is a history constructed by appending responses to some (possibly zero) pending invocations of H. The history $complete(H)$ is the subsequence of H consisting of all matching invocations and responses.

For a thread A, the *thread subhistory*, $H|A$ is the subsequence of events in H whose thread names are A. For an object x, the *object subhistory* $H|x$ is similarly defined. Histories H and H' are *equivalent* if for every thread A, $H|A = H'|A$. A thread or object history is *well-formed* if every response has an earlier matching invocation.

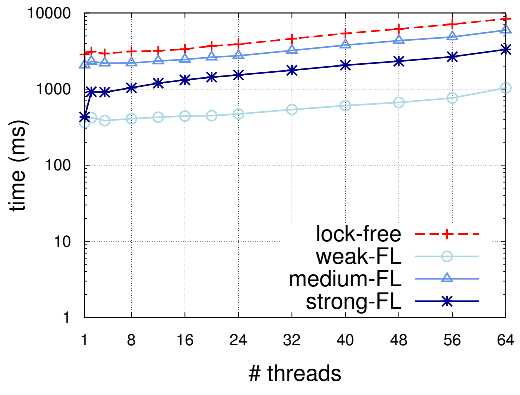
A method call m_0 *precedes* a method call m_1 in history H if m_0 finished before m_1 started: that is, m_0's response event occurs before m_1's invocation event in H. Precedence defines a *partial order* on the method calls of H: $m_0 \prec_H m_1$.



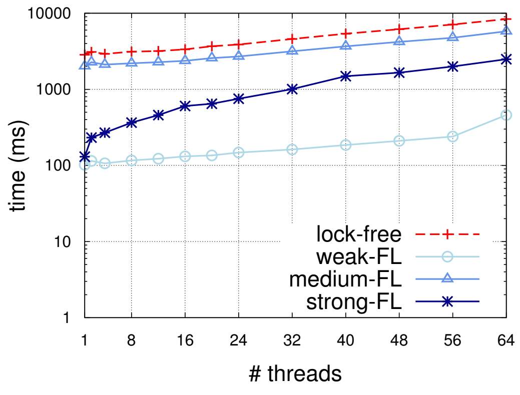
(a) slack = 1



(b) slack = 10



(c) slack = 20



(d) slack = 100

Figure 6: Linked list benchmark performance

A history H is *sequential* if the first event of H is an invocation, and each invocation, except possibly the last, is immediately followed by a matching response. If S is a sequential history, then $\prec_S$ is a total order. A *sequential specification* for an object x is a prefix-closed set of sequential histories called the *legal* histories for x. A sequential history H is *legal* if each object subhistory $H|x$ is legal for x.

For a history H, a partial order $\prec$ *extends* $\prec_H$ if for method calls m_0, m_1 of H, $m_0 \prec_H m_1$ implies $m_0 \prec m_1$, but not necessarily vice-versa. (Informally, different choices for $\prec$ will correspond to strong, medium, and weak futures linearizability.)

Definition 6.1. *For history H, and partial order $\prec$ extending $\prec_H$, H is* $\prec$-linearizable *if it has an extension H' and there is a legal sequential history S, with no pending invocations, such that*

L1 *complete(H') is equivalent to S, and*

L2 *if method call $m_0 \prec_{H'} m_1$, then $m_0 \prec_S m_1$ in S.*

We refer to S as a $\prec$-*linearization* of H.

Definition 6.1 generalizes the usual notion of linearizability in two ways: first, we do not require thread histories to be sequential, and second, we require that $\prec_S$ extends $\prec$ as well as $\prec_H$, a stronger requirement than the linearizability condition that $\prec_S$ extends $\prec_H$. We will use the partial order $\prec$ to capture the additional constraints on method ordering imposed by the various futures linearizability conditions.

6.2 Properties

A method is *total* if it is defined for every object state: for example, a `deq()` method that throws an exception on an empty queue. We now show that $\prec$-linearizability by itself never forces a thread with a pending invocation of a total method to block.

Theorem 6.2. *Let $inv(m)$ be an invocation of a total method. If $\langle x\ inv\ P\rangle$ is a pending invocation in a $\prec$-linearizable history H, then there exists a response $\langle x\ res\ P\rangle$ such that $H \cdot \langle x\ res\ P\rangle$ is also $\prec$-linearizable.*

Proof. Let S be any $\prec$-linearization of H. If S includes a response $\langle x\ res\ P\rangle$ to $\langle x\ inv\ P\rangle$, we are done, since S is also a $\prec$-linearization of $H \cdot \langle x\ res\ P\rangle$. Otherwise, $\langle x\ inv\ P\rangle$ does not appear in S either, since $\prec$-linearizations, by definition, include no pending invocations. Because the method is total, there exists a response $\langle x\ res\ P\rangle$ such that

$$S' = S \cdot \langle x\ inv\ P\rangle \cdot \langle x\ res\ P\rangle$$

is legal. S', however, is a $\prec$-linearization of $H \cdot \langle x\ res\ P\rangle$, and hence is also a $\prec$-linearization of H. □

We have just shown that like linearizability, $\prec$-linearizability is *non-blocking*: a pending invocation of a total method is never required to wait for another pending invocation to complete.

Also like linearizability, $\prec$-linearizability is *compositional*:

THEOREM 6.3. *H is $\prec$-linearizable if, and only if, for each object x, $H|x$ is $\prec$-linearizable.*

PROOF. The "only if" part is immediate from the definitions.

For each object x, pick a $\prec$-linearization H_x of $H|x$. Let R_x be the set of responses appended to $H|x$ to construct that $\prec$-linearization, and let $\prec_x$ be the resulting total order on method calls of x. Let H' be the history constructed by appending to H each response in R_x, for all objects x, in any order.

To show that H is $\prec$-linearizable, we argue by induction on the number of method calls in H'. For the base case, if H' contains only one method call, then it is sequential, thus trivially $\prec$-linearizable. Otherwise, assume the claim for every H' containing fewer than $k > 1$ method calls. For some object x, the last method call m in $H'|x$ must be maximal with respect to $\prec$: that is, there is no m' such that $m \prec m'$. Let G' be the history defined by removing m from H'. Because m is maximal with respect to $\prec$, H' is equivalent to $G' \cdot m$. By the induction hypothesis, G' is $\prec$-linearizable to a sequential history S', and both H' and H are $\prec$-linearizable to $S' \cdot m$. $\square$

A compositional property is sometimes called *local* [11].

6.3 Extension to Futures

For simplicity, we assume each future is evaluated at most once by the thread that created it. For each method m that returns a Future<T> object, we assume there is a corresponding method $\tilde{m}$ that returns a T object, and that the meaning of m is given by the object's sequential specification.

We model futures through simple re-writing rules. Let H be a history. Each Future<T> object in H is associated with two method calls: a first call m that creates the future, and a second call that evaluates it, both executed by a thread A. We will rewrite each such pair as a single call $\tilde{m}$ that returns a value of type T, yielding a new history $\tilde{H}$. While the method call precedence order $\prec_H$ is assumed to be total on $H|A$, the rewritten precedence order $\prec_{\tilde{H}}$ is not necessarily total on $\tilde{H}|A$. We then specify a partial order $\prec_F$ extending $\prec_{\tilde{H}}$, and require that $\tilde{H}$ be $\prec_F$-linearizable.

The exact rewriting rule depends on whether we are considering strong, medium, or weak futures linearizability.

Strong Futures Linearizability.

Construct $\tilde{H}$ by replacing m with $\tilde{m}$, treating the future object as a regular variable, and the future evaluation step as a simple assignment.

DEFINITION 6.4. *H is* strong futures linearizable *if $\tilde{H}$ is $\prec_{\tilde{H}}$-linearizable.*

Weak Futures Linearizability.

Construct $\hat{H}$ by replacing the invocation step of m with the invocation step of $\tilde{m}$, deleting the response step of m, deleting the future evaluation's invocation step, and replacing the future evaluation's response step with the response step of $\tilde{m}$.

DEFINITION 6.5. *H is* weak futures linearizable *if $\hat{H}$ is $\prec_{\hat{H}}$-linearizable.*

Medium Futures Linearizability.

Construct $\hat{H}$ just as for weak futures linearizability. If m_0 and m_1 are future creation calls in H, by the same thread on the same object, such that $\tilde{m_0}$ and $\tilde{m_1}$ are overlapping in $\hat{H}$, then they are unordered by weak futures linearizability, but should be ordered by medium futures linearizability. We define $\prec_m$ by strengthening $\prec_{\hat{H}}$ to order such calls.

$$\tilde{m_0} \prec_m \tilde{m_1} \text{ if } \begin{cases} \tilde{m_0} \prec_{\tilde{H}} \tilde{m_1}, \text{ or} \\ \tilde{m_0}, \tilde{m_1} \in \tilde{H}|A \cap \tilde{H}|x \text{ and } m_0 \prec_H m_1. \end{cases}$$

DEFINITION 6.6. *H is* medium futures linearizable *if $\hat{H}$ is $\prec_m$-linearizable.*

COROLLARY 6.7. *Strong, medium, and weak futures linearizability are all non-blocking.*

COROLLARY 6.8. *Strong, medium, and weak futures linearizability are all compositional.*

7. RELATED WORK

Halstead [5] originally proposed futures as benign annotations for side-effect-free computations. These futures were untyped and implicit: any object reference could be a future, and a run-time check was needed on each dereference. Modern futures are usually typed, and future creation and evaluation are explicit.

Liskov and Shrira [13] propose the notions of *call-streams* and *promises*. A call-stream allows one thread to make a sequence of remote procedure calls to another. Each such call returns immediately with a *promise* object, a kind of typed future that can be evaluated later to get the call's results. In our terminology, call-streams and promises satisfy medium futures linearizability, except with respect to streams instead of objects. Call-streams are intended for remote procedure calls, and make use of batching to reduce communication costs. Our treatment here is intended for shared objects in multicore systems, and utilizes type-specific optimizations, e.g., combining and elimination.

As noted, futures were originally proposed for side-effect-free computations. Welc et al. [19] consider the question of how futures should behave in languages like Java, where side-effects are common. They propose that futures should satisfy strong futures linearizability, and describe how to extend a JVM to support a speculative implementation of futures satisfying this property. Their futures implementation is general-purpose, and does not consider type-specific optimizations such as combining and elimination.

Scherer and Scott [16] propose *dual data structures*, objects whose methods are split into a request method, that registers the invocation and returns a ticket, and a follow-up method that takes the ticket as an argument, and returns either the request's response or a "not ready" indicator (for example, if trying to dequeue from an empty queue). If we interpret the ticket returned by a request as a kind of future, and the matching successful follow-up as that future's evaluation, then dual data structures satisfy weak futures linearizability. Note that the motivation for dual data structures is to provide a new way to implement linearizable partial methods, which is distinct from our motivation of using futures as a means of deploying highly-concurrent, type-specific techniques for scalable shared-memory data structures.

Futures provide a systematic and clean way to encapsulate a variety of optimization techniques, including *elimination* [8, 15, 17], where operations cancel one another without synchronizing at the object, *combining* [3, 4, 9] and *flat combining* [7], where a single thread executes method calls on behalf of multiple concurrent threads.

8. DISCUSSION

We have considered the use of futures in a novel context of long-lived shared data structures. We have shown that, in this context, futures enable multiple efficient optimizations, such as combining, elimination, flexible evaluation scheduling, contention reduction, etc. To make effective use of futures and to define which optimizations are permitted, we have proposed several correctness conditions, each extending linearizability. We have constructed three standard data structures – queues, stacks and linked lists – satisfying different correctness conditions and compared their performance with well-known lock-free alternatives. Our experience shows that in most cases the versions of the data structures satisfying the weakest correctness condition provide significantly better performance, up to two orders of magnitude. Furthermore, even the versions satisfying the strongest correctness condition substantially outperform the lock-free alternatives in certain cases.

Finally, a word about future work. We have not exhausted the optimizations permitted by futures. For example, our implementation of the linked list under medium futures linearizability does not allow a thread that issues operations such as inserting 3 and then inserting 2 to reorder those operations, because if it did, another thread might observe 2 but not 3 in the list. This danger could be averted, and the operations reordered, if the thread were to lock the shared list and apply multiple operations in a kind of atomic transaction. A promising area of future work is to determine whether such transaction-based approaches are scalable.

9. ACKNOWLEDGMENTS

We are grateful to Tim Harris, Yossi Lev, Mark Moir and anonymous reviewers for valuable feedback.

10. REFERENCES

[1] Bernstein, P., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems.* Addison Wesley Publishing Company, 1987.

[2] Calciu, I., Dice, D., Harris, T., Herlihy, M., Kogan, A., Marathe, V. J., and Moir, M. Message passing or shared memory: Evaluating the delegation abstraction for multicores. In *Proceedings of OPODIS* (2013), pp. 83–97.

[3] Goodman, J. R., Vernon, M. K., and Woest, P. J. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (1989), pp. 64–75.

[4] Gottlieb, A., Lubachevsky, B. D., and Rudolph, L. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *ACM Trans. Program. Lang. Syst. 5*, 2 (Apr. 1983), 164–189.

[5] Halstead, R. H. Multilisp: A Language for Concurrent Symbolic Computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (Oct. 1985), 501–538.

[6] Harris, T. L. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the International Conference on Distributed Computing (DISC)* (2001), pp. 300–314.

[7] Hendler, D., Incze, I., Shavit, N., and Tzafrir, M. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of ACM SPAA* (2010), pp. 355–364.

[8] Hendler, D., Shavit, N., and Yerushalmi, L. A scalable lock-free stack algorithm. *J. Parallel Distrib. Comput. 70*, 1 (2010), 1–12.

[9] Herlihy, M., Lim, B.-H., and Shavit, N. Scalable concurrent counting. *ACM Trans. Comput. Syst. 13*, 4 (Nov. 1995), 343–364.

[10] Herlihy, M., and Shavit, N. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[11] Herlihy, M. P., and Wing, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst. 12*, 3 (July 1990), 463–492.

[12] Lamport, L. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers C-28*, 9 (Aug. 1979), 690–691.

[13] Liskov, B., and Shrira, L. Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (1988), pp. 260–267.

[14] Michael, M. M., and Scott, M. L. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (PODC)* (1996), pp. 267–275.

[15] Moir, M., Nussbaum, D., Shalev, O., and Shavit, N. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of ACM SPAA* (2005), pp. 253–262.

[16] Scherer, W. N., and Scott, M. L. Nonblocking concurrent data structures with condition synchronization. In *Proceedings of DISC* (2004), pp. 174–187.

[17] Shavit, N., and Touitou, D. Elimination trees and the construction of pools and stacks: Preliminary version. In *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures (SPAA)* (1995), pp. 54–63.

[18] Treiber, R. *Systems Programming: Coping with Parallelism.* Technical Report RJ 5118. IBM Almaden Research Center, 1986.

[19] Welc, A., Jagannathan, S., and Hosking, A. Safe futures for java. In *Proceedings of the ACM Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2005), pp. 439–453.