Non-Blocking Inter-Partition Communication with Wait-Free Pair Transactions

Ethan Blanton[1] Lukasz Ziarek[1,2]

Fiji Systems Inc.[1] SUNY Buffalo[2]

{elb, luke}@fiji-systems.com

ABSTRACT

Predictable concurrency control is difficult. In this paper we present wait-free pair transactions, a lightweight, transactional communication object with the goal of achieving predictable communication between concurrent threads of execution, a key component to predictable concurrency control. Wait-free pair transactions allow entirely non-blocking, but one-way, communication between a dedicated reader and writer. Wait-free pair transaction objects provide transactional semantics for data consistency on the object being communicated by the reader and writer, but never require blocking, have a strictly bounded spatial overhead, and constant time overhead for any field accesses. We provide a detailed description of the implementation of wait-free pair transactions in Fiji VM and show how they can be leveraged for safe and predictable communication in a mixed-criticality environment. We demonstrate the runtime characteristics and predictability of wait-free pair transactions on a mixed criticality UAV flight control system benchmark.

1. INTRODUCTION

Reads from and writes to shared data are the most fundamental form of communication between threads in many real-time systems. Sharing data between threads that run concurrently requires a protocol to ensure that each thread accesses a consistent view of the data, changes to the data are not lost, and race conditions do not occur. In Java, these assurances are typically achieved via mutual exclusion (*e.g.*, monitors) — ensuring that two threads do not attempt to modify the same logical data structures at the same time, and that a thread does not attempt to view a data structure while it is in an inconsistent state due to ongoing modifications by another thread. We can view this as the *protocol* provided by the monitors that protect synchronized methods and blocks. Mutual exclusion presents difficulties for real-time systems, as shared data may be accessed by threads of differing priorities. Practical implementations require additional protocols to prevent or bound *priority inversion*, the situation where a lower priority thread holds a monitor required by a higher priority thread (thus blocking it), as well as careful design to ensure that critical sections in lower priority threads are not long enough to cause deadline misses in higher priority threads even under worst-case scenarios. Other techniques for achieving consistency include atomic updates and software memory transactions [17, 8, 18]. Atomic updates can generally be used only for very small data structures (a few machine words), and require hardware assistance to avoid falling back to a mutual exclusion implementation (*i.e.*, being backed by a lock, either at the OS or VM level). Transactions allow multiple threads to access shared data in a way that is logically equivalent to a serial schedule (*e.g.*, a schedule with mutual exclusion). Transactions can avoid enforcing strict blocking for updates, provided they can safely proceed in parallel. Nonetheless, they still require blocking or additional runtime constructs to resolve conflicting updates and out of order reads. Predictable unrolling of software transactions is an open problem, and one proposed solution is to fall back on mutual exclusion [23] — with all of the considerations thereof.

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
JTRES'13, October 09 - 11 2013, Karlsruhe, Germany
Copyright 2013 ACM 978-1-4503-2166-2/13/10 ...$15.00.
http://dx.doi.org/10.1145/2512989.2512994.

Synchronization and shared-state communication between Java VMs in a mixed-criticality multi-VM system is an open problem. The Java standard does not provide monitors between VMs, and monitors between criticality levels of a mixed-criticality system (with associated blocking) are problematic in general.

In this paper, we introduce *wait-free pair transactions (WFPTs)*, a lightweight transactional communication object that allows entirely non-blocking, but *one-way*, communication between a reader and a writer. Wait-free pair transaction objects provide transactional semantics for data consistency between the reader and writer, but never require blocking, have a strictly bounded spatial overhead, and impose constant time overhead on any object accesses. These properties are provided in return for the restriction that there can be only one reader and one writer for each WFPT. We can thus view a WFPT as a *one-way* predictable communication channel[1].

We provide two different execution models for WFPTs: 1) communication through WFPTs by local threads executing in a single virtual machine (VM) instance, and 2) communication across multiple VMs instances within the context of a multi-VM, to provide a *mixed-criticality* communication mechanism. The semantics of this mechanism ensure that communication across the partitions between virtual machines cannot violate priority constraints, cannot affect timings between virtual machines, and cannot lead to referential inconsistencies. To the best of our knowledge, wait-free pair transactions are the first mixed-criticality aware, direct, and predictable communication mechanism.

Specifically the contributions of this paper are as follows:

1. The design of a new, predictable communication mechanism for real-time systems called wait-free pair transactions. WFPTs provide consistency without requiring blocking and have

[1]We observe that two way channels can be encoded by leveraging multiple one way channels.

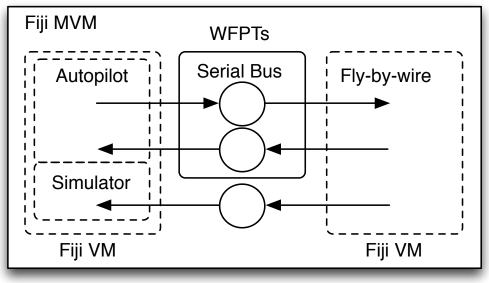


Figure 1: The structure of papabench split into multiple VM instances within the Fiji multi-VM. WFPTs are used as natural communication mechanisms between the VMs.

constant time overheads with bounded spatial constraints.

2. A prototype implementation of WFPTs in the context of Fiji VM. Our implementation supports WFPTs within a single VM as well as across VMs in a multi-VM deployment.

3. A performance evaluation and empirical validation of WFPTs on a UAV flight system benchmark on a multi-VM.

The remainder of this paper is structured as follows. We present a motivating example, a mixed-criticality flight control system, to illustrate the challenges of cross-partition predictable communication in Section 2 and explore the application of WFPTs to a multi-VM application. Section 3 describes the semantics of wait-free pair transaction objects. Section 4 presents the implementation of WFPTs within the Fiji VM and compiler. Results and evaluation are presented in Section 6. Related work and conclusions are given in Section 7 and Section 8, respectively.

2. MOTIVATING EXAMPLE

As a motivating example, consider jPapaBench [9], a complex aerospace benchmark written for Java. jPapaBench is derived from a core set of the Paparazzi [1] project, an open-source UAV flight system. To illustrate the benefits of WFPTs and their applicability to real-time systems, we focus on the communication mechanisms within the benchmark itself.

The jPapaBench code is conceptually divided into three major modules: the autopilot, which controls UAV flight and is capable of automatic flight in the absence of other control; the fly-by-wire, which handles radio commands from a controlling station and passes information to the autopilot to be integrated into flight control; and the simulator, which collects information from each of the other modules and determines the UAV's location, trajectory, and generates input from the environment (such as GPS data, servo feedback, *etc.*). Two of these modules, the autopilot and fly-by-wire (FBW), are housed in different microcontrollers on the conceptual hardware, and the jPapaBench code simulates a serial bus between them — they have no other direct communication paths. The simulator is only loosely coupled to the FBW module, but shares a moderate amount of state with the autopilot.

The autopilot and FBW module are logical candidates for division into a multi-VM system, as the implementation they simulate allows the two devices to proceed independently on separate microcontrollers. Therefore, we have divided the jPapaBench code into two virtual machines: the autopilot and simulator on one VM (as disentangling the simulator and autopilot is rather complicated and subtle), and the fly-by-wire unit on another. Communication between these VMs, simulating the serial interconnect bus, is accomplished via two wait-free pair transactions — one for each direction of bus communication. There is an additional WFPT providing fly-by-wire command state to the simulator. Fig. 1 depicts the split of core jPapaBench functionality into a multi-VM context.

3. WAIT-FREE PAIR TRANSACTIONS

In this section we outline the semantic properties of wait-free pair transactions and their runtime complexity in terms of high-level structures and operations. We provide encodings of these structures and operations in the following section. At its core each WFPT has a *reader* context and a *writer* context that correspond to the reader and writer threads of execution, respectively. For WFPTs used for communication within a VM, these contexts are Java threads or similar objects (*e.g.* RTSJ aperiodic event handlers). Each context has a different set of capabilities that together form a communication protocol. This protocol provides non-blocking and safe communication between the contexts. The essential capabilities of each context are provided by two primitive operations on the WFPT object, *commit* and *update*. WFPTs provide their guarantees through the use of object replicas and a precise protocol used to manage the replicas. We first discus the capabilities of the reader and writer and then show at a high level how these capabilities are provided by the replicas.

A writer on a wait-free pair transaction has the following capabilities:

- The writer can both read from and write to the object being communicated by the WFPT.

- The writer never loses coherence — anything previously written will be seen by the writer in subsequent reads until that value is overwritten by the writer itself.

- The writer never blocks on the reader, regardless of the state of the reader.

- The writer can *commit* its changes to the object stored in the WFPT at any time, causing them to become visible to the reader the next time the reader requests an *update*.

- Changes to the object stored in the WFPT will never be visible to the reader unless the changes are subsequently committed.

- The writer can read and write to the object stored in the WFPT in $O(1)$ time, regardless of concurrent activity by the reader.

A reader on a wait-free pair transaction, on the other hand, has the following capabilities:

- The reader can read from the objected bring communicated by the WFPT.

- The reader can also write to the object stored in the WFPT, but the writes will be lost when it performs an *update*. None of the reader's writes ever become visible to the writer.

- The reader never loses coherence unless it does an *update*; in the absence of updates, any value previously written will be seen in subsequent reads until that value is overwritten by the reader itself.

```
public class WaitFreePairTransaction {
    public WaitFreePairTransaction() { }

    public final void commit() { }
    public final void update() { }

    public final void setReader(Object reader) { }
    public final void setWriter(Object writer) { }
}
```

Listing 1: The wait-free pair transaction base class

- The reader does not see updates from the writer unless the writer does a *commit* and the reader does a subsequent *update*.
- The reader never blocks on the writer, regardless of the state of the writer.
- All reads and writes to the object stored in the WFPT are $O(1)$ regardless of concurrent activity in the writer.

The reader context is allowed to modify the object stored in the WFPT to perform local computations on the object itself without necessitating a copy of that object. Notice that the writer context cannot invalidate this object unexpectedly as the reader is required to perform an *update* to receive the writer's most up to date changes. As such, the reader controls when its copy is updated. This has two benefits: the aforementioned safety and consistency, as well as not requiring pervasive changes to a system that uses WFPTs. A reader and any computation that it performs does not need to create and manage copies of the objects communicated through the WFPT.

As these capabilities suggest, the interface to the conceptual transaction provided by the WFPT is given through low-level calls to *commit* and *update* as well as a special object representation for WFPT objects. Every WFPT object extends the `WaitFreePairTransaction` base class, shown in Listing 1. These objects are treated specially by the compiler and runtime. WFPT objects logically contain four versions of the object that the reader and writer contexts manipulate, which we call replicas, along with some tracking information. Appropriate manipulations of these replicas and information are used at runtime to provide transactional semantics.

Compiler and VM support for the special object representation used by WFPT objects is required to achieve transparent field access with WFPT guarantees. The purely transactional semantics of WFPT objects can be provided without compiler and runtime support, provided that the reader and writer contexts manage the object replicas explicitly. However, this approach is much more involved and error-prone than compiler- and runtime-assisted WFPT objects.

3.1 Tracking Structures

To achieve non-blocking communication between the contexts in a predictable fashion, WFPT objects use replication. The WFPT structure is shown in Fig. 2. Each WFPT object contains four distinct replicas, labeled as W, F, U, and R in the figure. Thus, WFPTs will operate over four copies of the data being communicated between contexts. These replicas are created when the WFPT object is allocated.

The WFPT tracking structure maintains four logical indices for each WFPT replica, one corresponding to each replica created at initialization time, as well as an update flag indicating whether the writer has committed a version not yet updated by the reader. These

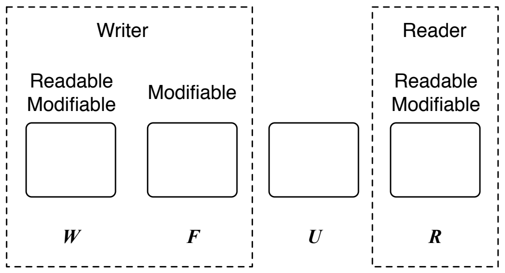


Figure 2: The structure of a wait-free pair transaction consisting of four replicas W, F, U, and R.

	Writer		Reader	
	Obj.	**Ind.**	**Obj.**	**Ind.**
W	RM	R	-	-
F	M	RM	-	-
U	-	RM	-	RM
R	-	-	RM	RM

Table 1: Access rights allowed to each context on the four object copies and their indices. See text.

indices are used to identify which objects are safe to modify and read by the two contexts utilizing the WFPT. The four indices are:

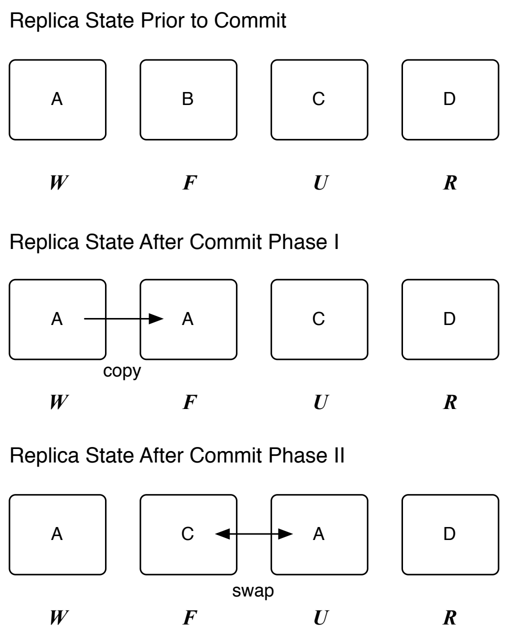
1. W: The version currently being modified by the writer,
2. F: The version that will receive the writer's next commit,
3. U: The version the reader will read from after it performs an update, and
4. R: The version currently being read by the reader.

Only the writer can read or modify the objects at indices W and F, and only the reader can read or modify the object at index R. Neither context can read or modify the object at index U. Table 1 lists the access rights that each context has for the object at each index, as well as the index itself. The letter 'R' is used to indicate read access, 'M' modify access, and '-' indicates that the specified context has no rights on a given object or index. The key observation is that the object replica at the index U is *not* accessible to either the reader or writer contexts. This replica provides a bridge between the reader and writer contexts and is key to providing data consistency as it guarantees both data consistency and repeatable reads. The other indices provide non-blocking guarantees as well as the time complexity guarantees for WFPTs.

3.2 Performing Commits and Updates

Careful reassignment of the four indices maintained in the WFPT tracking structure is used to encode the *commit* and *update* operations. Rules about which context can modify which indices ensure $O(1)$ manipulation of the index fields. These two basic operations are detailed below and shown graphically in Fig. 3 and Fig. 4.

The *commit* operation is performed by copying the contents of the object at index W to the object at index F, and then atomically swapping the indices of F and U and setting the update flag. Because only the writer can access the object at F, the object copy is not visible to the reader. Because neither context can access the object at U, the committed object is not visible to the reader until it performs an update. After this operation, the object at F is ready to



Replica State Prior to Commit
A
B
C
D
W
F
U
R
Replica State After Commit Phase I
A
A
C
D
copy
W
F
U
R
Replica State After Commit Phase II
A
C
A
D
swap
W
F
U
R

Figure 3: The phases of a wait-free pair transaction commit operation. In the first phase the committed objects is copied from *W* to *F*. In the second phase the objects in *F* and *U* will be swapped atomically.

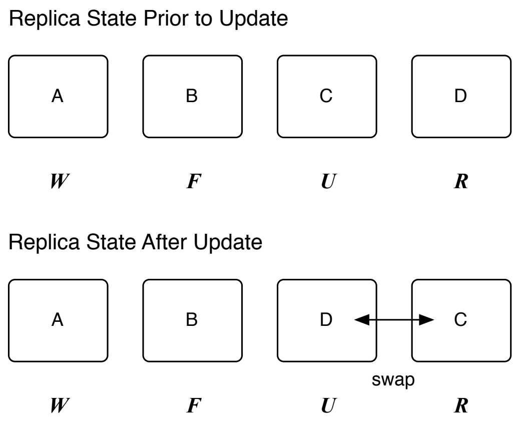


Figure 4: The phases of a wait-free pair transaction update operation. In the first phase the reader check to see if the update flag has been set. In the second phase the reader swaps the objects in *R* and *U*.

receive the next commit by the writer, and the object at *U* is ready to become the read object when the reader performs and update.

The *update* operation is similar, but comparatively simpler. The reader first checks the update flag; if it is not set, no action is taken. If it is set, the update is performed by atomically swapping the values of *R* and *U* and clearing the update flag. The safety of this operation is guaranteed by the fact that the writer cannot access either *R* or *U*, similar to the writer's manipulations above. After this operation, the object at *R* represents the writer's most recent commit and the object at *U* is stale and will not be used until the writer commits and it is moved to *F*.

3.3 Accessing WFPT Fields

Access (both reading and writing) to fields of the WFPT must use the appropriate copy. In order to accomplish this, the compiler must insert read and write barriers for WFPT fields. These barriers use the WFPT tracking structure to determine whether the current context is a reader or writer, and look up the appropriate copy via the *R* or *W* index as appropriate. Because `WaitFreePairTransaction` is a base class, and not an interface, this check needs only be performed for objects that are known at compile time to be WFPTs, incurring no extra cost for non-WFPT objects.

Static fields on a WFPT object are not protected by WFPT semantics. The strict one-way communication and one-to-one context relationship limit the usefulness of static fields for communication. Rather than tackle the added complexity and restrictions involved with ensuring the integrity of static fields in the face of multiple object instantiations with potentially dissimilar read and write contexts, static fields are simply removed from consideration.

3.4 WFPT Overhead

For the higher-priority context on a WFPT, every operation except *update* is constant-time, including field accesses. An *update* is linear in the size of the WFPT object, due to the necessary copy from the object at *W* to the object at *F* (as described in Section 3.2). Moreover, all field and method accesses proceed in time *completely independent* of the lower-priority context. The atomic operations necessary for index updates may cause *commit* and *update* to experience tightly bounded blocking, depending on implementation details, but this can be avoided through the use of hardware-assisted compare-and-swap (CAS) operations.

4. IMPLEMENTING WFPTS

We have implemented wait-free pair transactions in the Fiji VM [13] and associated compiler. The implementation provides all of the semantics described in Section 3, including non-blocking *commit* and *update* using hardware CAS. Modifications to the compiler are minimal and non-invasive, and modifications to the runtime are limited to a single check in reflective object instantiation and two new classes (one empty base class and one helper class to store the tracking information for each WFPT object).

Instantiations of `WaitFreePairTransaction` objects are represented at runtime by their tracking object, with all references to the WFPT object replaced by the tracking object, called `WFPTProxy`, after initialization. This necessitates insertion of some extra compiler barriers (e.g., for type casts) but simplifies other operations and requires no extra fields to be stored on the user's `WaitFreePairTransaction` object.

The tracking object structure is given in Fig. 5 and consists of an index word and a replica array. Not shown are fields to store the context identifiers (*e.g.*, `Thread` references), as their use is uninteresting. The replica array conceptually stores the replicated WFPT object. The index array encodes indices to the *W*, *F*, *U*, and *R* replicas within the replica array, as well as the update flag. To be able to perform one CAS to swap indices, all WFPT metadata must be encoded into one word. This packing is achieved by placing all four indices and the update flag in a single index word. As the range of each index is only 0–3, two bits will suffice for each index (for a total of eight bits), and the update flag requires only one extra bit.

4.1 Compiler Changes

Instrumentation was added to the Fiji VM compiler to insert various barriers and instrumentation in the generated code to provide WFPT semantics. These fall broadly into four categories:

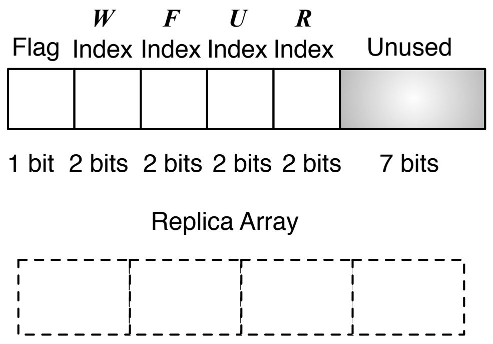


Figure 5: The low level structure of a wait-free pair transaction consisting of and index word encoding the update flag and indices to the four replicas *W*, *F*, *U*, and *R*. The four replicas are stored in a replica array.

- *Tracking structure allocation* to ensure that a WFPT tracking structure, `WFPTProxy`, is created and initialized for each instantiation of a WFPT object and used in its place,
- *Method lookasides* to replace the empty methods on the WFPT base class with their implementations,
- *Cast replacements* to handle casting of `WFPTProxy` object references as the objects they represent, and
- *Access barriers* to perform reader/writer context lookups and redirect field accesses to the appropriate object replica.

These modifications are less than 500 lines (excluding documentation) of almost entirely self-contained compiler code. No changes are required to the existing compiler code aside from a small number of convenience references in a helper class and an explicit liveness determination for the `WFPTProxy` class (preventing it from being prematurely removed from the build) if wait-free pair transactions are in use.

Tracking structure allocation.

In order to allocate the tracking structure and object replicas, the NEW and INVOKESPECIAL Java bytecodes corresponding to allocation of a WFPT object and invocation of its constructor are intercepted by the compiler. After constructor invocation, a `WFPTProxy` object is allocated and initialized. Initialization of this object includes allocation of memory for the object replicas and copying of the initialized object contents to the replicas at both the *W* and *R* indices. The LHS of the NEW bytecode is then artificially replaced with the tracking structure, so that all references to the WFPT object will in fact be references to the tracking structure.

Method lookasides.

The `WaitFreePairTransaction` class provides four final methods for manipulating WFPT state. However, for simplicity of implementation specific to the Fiji VM, these methods are *actually* provided by `WFPTProxy`. The compiler replaces the `WaitFreePairTransaction` method invocations with their `WFPTProxy` equivalents.

Cast replacements.

Because the object reference used for WFPT objects is actually a reference to an instance of `WFPTProxy`, casts to WFPT-derived types that cannot be statically eliminated must be handled specially at runtime. In addition, casts to interface types for which there exists at least one WFPT implementation must be handled similarly. The compiler inserts a cast barrier for these operations that first checks the object being cast to determine whether it is an instance of `WFPTProxy`, and then casts either the object itself or the object managed by the proxy as appropriate.

Access barriers.

The most frequent, and important, change emitted by the compiler is access barriers for WFPT object fields. These accesses require a lookup to determine whether the current context is a reader or a writer on the WFPT being accessed (or neither, in which case an exception is raised). Once the current context has been determined, the access can proceed on the replica indicated by either the tracking structure's *W* index or *R* index as appropriate. Because `WaitFreePairTransaction` is a base class, these barriers are inserted only for fields that are known, at compile time, to be WFPT fields.

4.2 Runtime Support

Most of the real work to provide WFPT guarantees occurs in the runtime via standard Java classes — indeed, compiler involvement is only required because transparent field accesses are not otherwise possible. Only one minor change is required in the existing runtime libraries. The existing VM-specific implementation for the Java `Class` class requires minor modification to the `newInstance(Class<?>)` method to ensure that tracking structures are created and properly initialized for reflective instantiation.

The WFPT tracking structure and various support methods are provided by the class `WFPTProxy`. This class implements the `commit`, `update`, `setReader`, and `setWriter` methods, as well as several methods to assist with type casts. The context management methods are straightforward, owing to an implementation restriction that forbids changing contexts after their initial configuration. This restriction eliminates the need for internal synchronization for the `setReader` and `setWriter` methods and simplifies their implementation. However, careful implementation of the *commit* and *update* operations is essential to avoiding blocking in the higher-priority context. We will consider each of these operations, and the necessary data structures, in turn.

Commit.

The `commit` method, implementing the *commit* operation on the WFPT, modifies the *F* object, the *U* and *F* indices, and the update flag as described in Section 3.2. Fig. 6 shows the two phases of the commit operation.

Because the reader context neither views the object at *F* nor modifies the index *F*, copying of the *W* object to the *F* object can proceed without regard to the reader's state and without using any sort of synchronization mechanism. Updating the indices, however, is an operation that is visible to the reader and must be performed either atomically or in a synchronized fashion. Careful placement of the indices and flag allows a simple one-word atomic compare-and-swap (CAS) operation to update both flag fields and the update flag, and *ensures that there is never a priority inversion, even bounded.*

A commit operation swaps the *F* and *U* fields in this word, sets the update flag bit, and performs a CAS against the unmodified value. Upon success, the commit is complete. Failure indicates that the (necessarily higher priority) reader context preempted the writer, and thus the *U* field may be dirty. Note that the *F* field is unaffected, so no object copy need be repeated, and that this

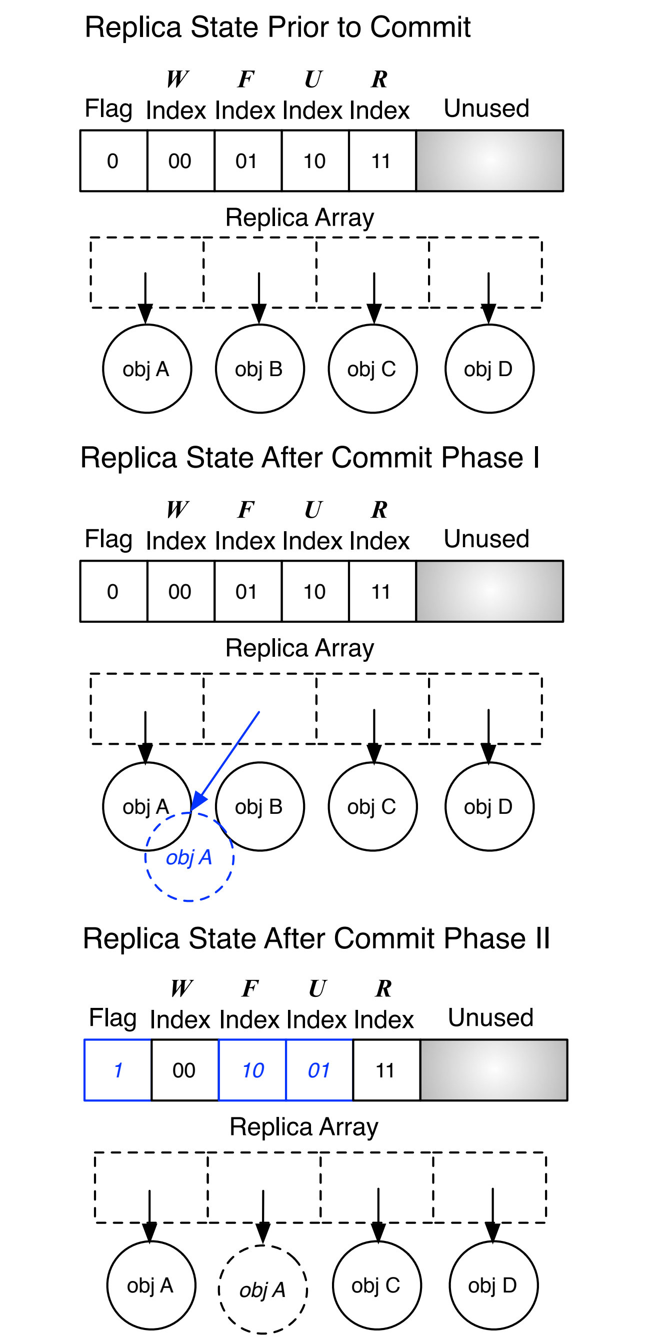

Replica State Prior to Commit
W F U R
Flag Index Index Index Index Unused
0 00 01 10 11
Replica Array
obj A obj B obj C obj D
Replica State After Commit Phase I
W F U R
Flag Index Index Index Index Unused
0 00 01 10 11
Replica Array
obj A obj B obj C obj D
obj A
Replica State After Commit Phase II
W F U R
Flag Index Index Index Index Unused
1 00 10 01 11
Replica Array
obj A obj A obj C obj D

Figure 6: The phases of a wait-free pair transaction commit operation at runtime. In the first phase the committed objects is copied from *W* to *F*. In the second phase the objects in *F* and *U* are swapped atomically.

situation can arise only if there is an outstanding commit that has not yet been updated (and thus the update flag bit is set). The writer context simply recomputes the index word with the new value for *U* and tries again until the CAS succeeds.

Update.

The `update` method is similar, but less complicated. Because no object copy is required for *update*, the reader context simply

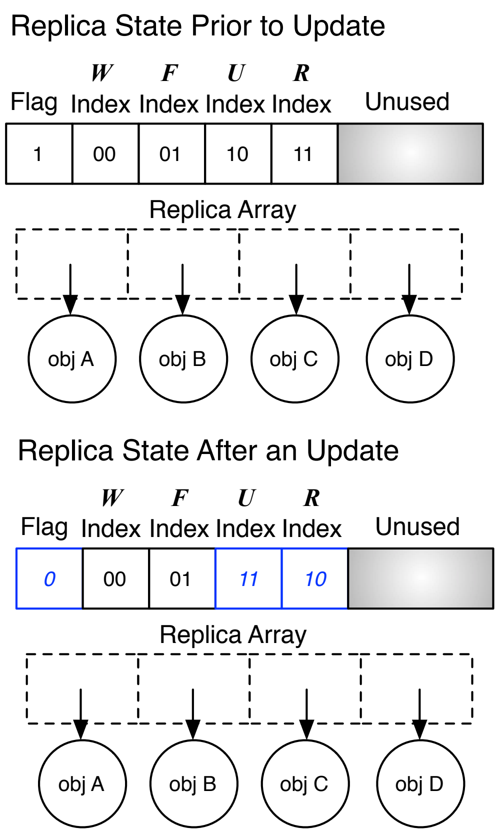

Replica State Prior to Update
W F U R
Flag Index Index Index Index Unused
1 00 01 10 11
Replica Array
obj A obj B obj C obj D
Replica State After an Update
W F U R
Flag Index Index Index Index Unused
0 00 01 11 10
Replica Array
obj A obj B obj C obj D

Figure 7: The phases of a wait-free pair transaction update operation. In the first phase the reader check to see if the update flag has been set. In the second phase the reader swaps the objects in *R* and *U*.

checks the update bit in the index word and, if it is set, swaps the *R* and *U* field, clears the update bit, and issues a CAS against the stored previous value of the index word as shown in Fig. 7. Success indicates that the update is complete, and failure indicates that there was a commit/update collision and the new index word must be recomputed and the CAS retried until it succeeds.

The existence of the fourth *F* field and the non-shared *U* field (and corresponding replicas) enable the guarantee that a successful *update* operation in this implementation will always leave the *R* index pointing to the previously committed, consistent object or the most recently committed, consistent object even in the face of a commit/update race. Because neither the reader nor writer context can access the object at *U* without first using a CAS to move it to *R* or *F*, respectively, the reader will never have access to an inconsistent object.

Predictability.

Predictability is directly related to the the collision rate of the commit and update operations. This rate itself depends on the rate of the individual commit and update operations performed by the writer and reader, respectively. Using strong CAS operations on a single-processor system, the lower priority context cannot cause a CAS failure in the higher priority context under the implementation described here. The higher priority context, however, can cause CAS failures (and thus retries) in the lower priority context. As a practical matter, the window for this is very narrow (a few machine instructions), so repeated failures are likely to be an indication of an erroneous application. In a well-behaving application, the higher priority context should succeed immediately on *commit* or *update*, and the lower priority should retry its CAS at most once.

Interactions with Automatic Memory Management.

The design of WFPTs makes them amenable to garbage collection as the WFPT object is managed as a single, albeit complex, object. No changes were necessary to support WFPTs in Fiji VM's GCs [12] for WFPT used within a single VM. Careful consideration, however, needs to be given for WFPTs that facilitate communication between partitions in a mixed-criticality system. Section 5.1 describes this in more detail.

4.3 Optimization of WFPT Accesses

The wait-free pair transaction implementation examined here is entirely unoptimized. In particular, context lookups are performed for every object field access. We anticipate that many such lookups could be elided by an optimizer in a manner similar to type checks, but we have not yet implemented this optimization. Optimized context lookups have the potential to reduce WFPT field access overhead to zero (compared to general object field accesses) for entire methods, loops, or call graphs. This optimization could also improve transaction performance, but in the general case we would expect field access to dominate transaction operations in number.

The semantics of WFPT objects provide an optimizer with, in some ways, an easier problem than type cast elision. Since a given thread can be only one of the reader or writer context on a WFPT object, and there is exactly one reader and one writer for any given WFPT object, repeated accesses to the same object in a call graph are ensured to be in the same context. Furthermore, the *update* and *commit* operations are performed by the VM infrastructure, and known to the compiler, so the compiler is aware of potential changes to the WFPT state for a given context. Implementing these optimizations, as well as other possible optimizations, is left for future work.

5. INTER-PARTITION WFPT

Wait-free pair transactions generalize readily to communication between partitions in a multi-VM or mixed-criticality system, if the reader and writer contexts are represented by virtual machines rather than threads. Their wait-free properties are particularly valuable for mixed-criticality settings, since the higher priority partition is guaranteed constant-time interactions with the lower-priority partition. Some extra precautions must be taken to ensure that partitioning guarantees are not violated. The semantic differences between intra-partition WFPT and inter-partition WFPT are:

- The *reader* and *writer* contexts are represented by VM instances. Any thread within a VM can access an inter-partition WFPT object according to the VM's context. The application must use synchronization if it requires consistency between accesses in the face of multiple threads accessing the same WFPT.

- Care must be taken to maintain spatial partitioning as well. In this work we solve that by allowing only *primitive objects* to serve as inter-partition WFPTs. A primitive object, in this context, is an object which has only primitive fields. This prohibits inter-partition WFPTs containing references or arrays, although this restriction could be relaxed for arrays of primitives under certain circumstances.

- Static fields are not only not subject to WFPT semantics, they are not shared between VMs. This restriction avoids potential inter-VM interactions related to static initializers.

5.1 Referential Integrity

References to objects visible in multiple VMs must be ensured referential integrity in each VM. The restriction to primitive objects for inter-partition WFPTs frees the infrastructure from tracking references stored in the WFPT, but references *to* the WFPT must still be protected. Our implementation handles this by storing inter-partition WFPTs in a memory scope (similar to those defined by the Real-Time Specification for Java [7] and Safety Critical Java [15]) that is maintained by the multi-VM infrastructure and *effectively immortal* from the point of view of both VMs.

The infrastructure must also provide a way to transport references between VMs in a safe fashion. In the Fiji VM implementation, VMs declare their intention to use an arbitrary number of inter-partition WFPT objects (limited by available memory), along with their type, the reading and writing VMs (by name), and an identifying number. The infrastructure allocates and initializes these objects in the shared immortal memory space, then each VM requests the objects it declared by an identifier assigned at declaration time.

5.2 Type System Reconciliation

Sharing type systems between virtual machines in a multi-VM has challenges in a mixed-criticality environment. These challenges range from tracking static initialization status across VMs in an efficient manner to adding classes without potentially triggering priority inversions between VMs. Rather than tackle all of these issues, we chose to extend the WFPT-specific type cast instrumentation in the compiler and runtime to handle cross-partition casts and references.

This approach requires a small number of straightforward restrictions:

- A shared inter-partition WFPT object must have the same fields in both VMs. Multi-VM infrastructure handles field copying, so they need not have identical object model or layout, but our preliminary implementation also requires this.

- The `WFPTProxy` class must have the same fields, object model, and field layout in both VMs. This object is provided by the multi-VM infrastructure, so this requirement is necessarily met.

- The `WFPTProxy` class type record must have the same in-memory layout in both VMs. This is also satisfied by providing its implementation in the multi-VM.

Object headers are carefully manipulated at WFPT creation time such that the inter-partition WFPT objects manipulated in each VM always present the type system header appropriate for the current context. This is accomplished by placing a writer VM type header on the object at *W* (which never moves) and a reader VM type header on the other three copies of the object. Since the multi-VM infrastructure handles copying between these objects, their type header mismatch can be handled there as necessary.

The `WFPTProxy` tracking structure (arbitrarily) carries a type header from the writer VM. The type casting infrastructure cannot therefore use the type system's fast path casting for `WFPTProxy` objects. Instead, casts involving this class consult the object's type record directly and compare its canonical name representation for equivalence. This simple cast technique is suitable only because `WFPTProxy` is final, system-provided, and directly accessed only in well-defined circumstances.

6. EVALUATION AND RESULTS

We evaluate WFPT performance and suitability for intra- and inter-VM communication using both micro-benchmarks and jPapaBench (see Section 2 for more information on jPapaBench). We evaluate our implementation of WFPTs on two hardware platforms: a Raspberry Pi and a LEON3 development board.

The Raspberry Pi represents a small, general purpose computing platform dedicated to a single real-time task. The system used in our experiments is a Model B revision 2, which has a 700 MHz ARM v6 processor and 512 MB of RAM running Linux 3.2.x. No software other than critical system daemons (e.g., `dhclient`, `udevd`, `sshd`) are running during the tests, and no other processes on the system are configured with real-time priorities. The test binaries run with real-time priority using the FIFO scheduler.

The second configuration is targeted at moderately resource constrained embedded systems. For this configuration we leveraged a stock LEON3 embedded board manufactured by Gaisler. The experiments were run on a GR-XC3S-1500 LEON development board [2] running RTEMS version 4.9.6. The board's Xilinx Spartan 6 Family FPGA was flashed with a LEON3 configuration running at 40Mhz. The development board has an 8MB flash PROM and 128MB of PC133 SDRAM. The test builds use the FIFO scheduler.

6.1 WFPT Overhead

Wait-free pair transactions incur no overhead for any operations other than WFPT field accesses, transaction operations, and checked casts to WFPT types or interfaces implemented by a WFPT. Because type casts are both infrequent and aggressively optimized in the Fiji VM, we consider only WFPT field access and transactions in our evaluation. These operations are compared to roughly equivalent monitor synchronized communication to establish a baseline for comparison and to characterize WFPT overheads. As mentioned in Section 4.3, no optimization of context lookup for WFPT field access has been performed in the current implementation.

The micro-benchmark for this evaluation considers a worst-case scenario for both monitors and wait-free pair transactions. For the monitor baseline benchmark, the test apparatus repeatedly calls the following method:

```
public synchronized void set(int value) {
    this.value = value;
}
```

This method incurs a monitor lock and unlock for every field store. Similarly, the WFPT benchmark apparatus repeatedly calls the following function in the writer, with or without an immediate commit:

```
public void set(int value) {
    this.value = value;
}
```

This incurs a WFPT context lookup for every field store. For experiments exercising commits, there is a commit immediately following invocation of this function on every iteration. Note that this method is identical to the method used for monitor evaluation, except without the `synchronized` keyword on the method.

Each micro-benchmark runs the aforementioned methods ten million times, timestamping before and after the sequence of invocations with nanosecond timestamps. The actual precision of the timestamp is dependent on the platform; on the Raspberry Pi, it is reported to be 1 ns[3], and on the LEON3 it is 100 μs. This trial is repeated ten times, and the results of the first trial are discarded.

[2]Additional board specification can be found at Gaisler's website: `http://www.gaisler.com/`.

[3]Our experiences with Linux have yet to produce such precision in the timing data. In this case, the data suggests approximately 1 μs effective timestamp precision.

Configuration	ARM	LEON3
Monitors	123	5107
WFPT	260	6557
WFPT w/ Commit	653	14107

Table 2: Mean per-iteration duration of each micro-benchmark configuration. All times are in nanoseconds.

Table 2 shows the mean per-iteration duration of three different micro-benchmark configurations on both platforms. The three configurations are: the monitor-guarded update, WFPT field update, and WFPT field update with commit. Note that the granularity of timings requires a large number of iterations to achieve meaningful results, so per-iteration jitter is not readily available without hardware assistance. However, jitter from *trial to trial* (trial durations are well over one minute) is low enough on the LEON3 that the 100 μs timer granularity becomes problematic. Jitter on the Raspberry Pi is larger, as is typical on non-RTOS systems.

These trials do not include contention (in the form of a second thread locking the monitor or a reader on the WFPT), as on these single-core processors there is not a good way to distinguish between time used by the contending thread versus the trial thread without significantly higher-precision timestamps. We did, however, perform tests with very sparse contention from a higher-priority thread to ensure that no pathological effects emerged, and no such effects were observed. Trial durations were slightly extended by contention for the processor, which is not surprising.

We do not consider reader access to WFPTs in this performance evaluation for practical reasons. First, field access follows the same code path between reader and writer. Second, the *update* operation is strictly less complex than the *commit* operation, as the CAS phase is equivalent and *update* does not require an object copy. Finally, *update* is short-circuited in the event that the update flag is not set, so timing an update operation requires a corresponding commit, and either timing a single update or disentangling the updates from commits in multiple iterations are problematic for the reasons discussed above.

Our conclusion from these results is that while WFPT object field access is slower than standard object field access with monitor guards, and WFPT transactions are slower than monitor interactions, the cost is moderate enough to justify the use of WFPTs where blocking for synchronization cannot be tolerated. For this worst-case scenario for both WFPTs and monitors, the cost of WFPT field access is only about twice that of a monitor-protected interaction on ARM (and far less on LEON3), and the cost of access plus the transaction itself is about six three times the monitor-protected interaction on ARM, and only three times on the LEON3. The unoptimized implementation of WFPTs will fall behind monitors as the number of field accesses is increased, but optimizations as discussed above should mitigate that to a great extent.

6.2 WFPTs in Multi-VM jPapaBench

We have implemented a multi-VM version of jPapaBench, as described in Section 2. The details of the division and the associated wait-free pair transaction objects are provided here, as well as some performance data.

As previously described, communication between the jPapaBench autopilot and fly-by-wire modules is via a simulated serial bus, and

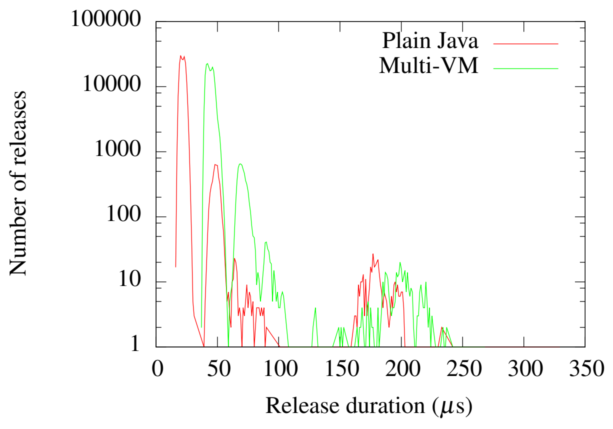


Figure 8: Multi-VM and plain Java jPapaBench release times for the Stabilization task. Release durations are rounded to the nearest microsecond.

our multi-VM implementation transports this communication via two inter-partition WFPT objects, one for each direction of communication. These two WFPTs are identical, carrying the fixed-format data record exchanged on the serial bus. This data consists of four bytes of status information and nine words of data received on the fly-by-wire module's radio interface and relayed to the autopilot. This mirrors the existing implementation of queue-based communication in the plain Java jPapaBench code. The WFPT carries an additional serial number, incremented by the writer prior to each *commit* and used by the reader to determine whether the writer has sent new information via the WFPT, and a boolean flag used for convenience in the reader and ignored by the writer.

A third wait-free pair transaction, communicating between the fly-by-wire module and the simulator, carries the raw radio data received by the fly-by-wire plus an extra boolean flag added so the reader can tell when new data has arrived. This flag is set by the writer on each *commit* and cleared by the reader after the data has been viewed. A successful *update* will therefore re-set the flag, letting the reader know that new data has arrived.

Each of these structures contains a small amount of metadata (the serial number on serial bus data packets, and the boolean flag on the radio commands structure) to help the reader determine whether a commit has occurred. This illustrates a generally useful technique for using wait-free pair transactions between VMs, where no standard Java mechanism exists for asynchronous notification of the commit. In the case of serial numbered commits, the reader simply notes the current serial number in a local variable, performs an update, and then compares the new number with the old. If serial numbers are sequentially assigned, this can also indicate whether commits were "lost" by the reader. Indication of whether a commit has occurred can be communicated via a return value on *update*, which is an improvement we will implement going forward.

Multi-VM jPapaBench was exercised via the flight plans distributed with the benchmark, and did not display any behaviors incompatible with the plain Java implementation. No missed deadlines were detected on either the Raspberry Pi or LEON3, and mission objectives were either achieved or failed as expected.

Fig. 8 shows a collection of releases of a single representative jPapaBench task in both the plain Java and multi-VM implementations. The selected task, the stabilization task in the autopilot module, communicates with the fly-by-wire module via the simulated serial bus, and thus uses an inter-partition WFPT in the multi-VM implementation. The jPapaBench plain Java and multi-VM implementations were run 10 times each, in alternation, on the Raspberry Pi[4] and the first run of each implementation was discarded. Release durations for the Stabilization task were aggregated across the nine remaining trials, rounded to the nearest 1 μs, and the frequency of each duration is plotted in the figure. From Fig. 8 it is clear that the multi-VM implementation increases the duration of each release by about 25 μs, but the distribution of durations is otherwise little affected. In this particular task, 25 μs is roughly the duration of the shortest plain Java releases, and three orders of magnitude faster than the period of this task. Other tasks in the system interacting with WFPTs display similar results and are not shown here.

This exercise of dividing jPapaBench into a multi-VM system demonstrates that inter-partition WFPTs are realistic for safe, predictable inter-VM communication with no blocking between VMs in a real Java application. The results suggest that practical performance of inter-partition WFPTs is suitable for moderately complex communication when non-blocking operation is desirable.

7. RELATED WORK

There has been a lot of work on real-time Java [3, 2, 4, 13], the Real Time Specification for Java (RTSJ) [7], and Safety Critical Java (SCJ) [15]. We believe wait-free pair transactions can be used by a Java VM wishing to provide a predictable communication mechanism. Specifically, we believe WFPT will be particularly useful for VMs supporting mixed-criticality execution models.

Recently software transactional memory [17, 8, 18] has emerged as an alternative concurrency control mechanism to locking. STM prevents common locking errors such as deadlock and has been utilized to prevent priority inversion [20]. Unfortunately, such techniques are inherently unpredictable since the underlying systems also utilize transactions for concurrency control resulting in aborts due to failed speculation. The most closely related is work on preemptible atomic regions, discussed in more detail below.

Priority rollback protocol (PRP) [21] has been proposed as a fast and predictable alternative to the priority inheritance protocol [6, 16] or priority ceiling protocol [5]. PRP relies on transactional tracking structures for all reads and write performed while holding a lock. High priority threads can safely acquire the lock, even if a low priority thread has not completed its critical section. Like WFPTs, PRP can be leveraged to implement communication protocols between threads. However, WFPTs never require unrolling and re-executing code and offer tighter predictability bounds at the cost of a less general interface.

Preemptible atomic regions, or PARs [10] are low-latency synchronization primitives for real-time systems and have been implemented in OVM [4]. PARs have shown better latency performance than traditional locking in substantial benchmarks derived from actual flight software. PARs allow for a critical region protected by a PAR to be reverted if a higher priority thread attempts to enter another critical region also protected by a PAR, in much the same fashion that a low priority thread executing within a critical region protected by a PRP lock can be rolled back. This ensures that a high priority thread can execute its critical region with extremely low latency. Unlike WFPTs, PARs introduce a new programming model requiring programmers to explicitly juggle locks and PARs.

WFPTs, unlike each of the foregoing techniques, focus only on predictable communication between threads and do not provide a general purpose concurrency control mechanism. Although we can envision building WFPT-like mechanisms from a more robust concurrency control mechanism like STM, PRP, or PARS, WFPTs

[4]Timing precision on the LEON3 did not allow meaningful results.

have no impact on the overall programming model and have modest runtime overheads due to their specialized nature.

There have been many mechanisms that support various types of message passing, including synchronous [14], asynchronous [11], and heterogeneous [22]. We expect to be able to encode higher-level message passing primitives with precise predictability guarantees using WFPTs.

7.1 Simpson's Four-Slot Mechanism

Simpson presents a *four-slot mechanism* [19] that shares the goals of wait-free pair transactions. Like WFPTs, the four-slot mechanism also uses four communication "slots" (which do not necessarily contain object replicas, but may) and two one-bit fields to implement a protocol that cooperatively determines which of the slots are currently in use by the reader or the writer. As presented, the four-slot mechanism is not directly suitable for WFPT semantics, as it provides *coherence* but not *persistence* for the write object — that is, the writer "loses" the state it wrote after each write. However, the *data visibility mechanism* represented in Section 4.3 of [19] could be used to provide the backing for a protocol with equivalent semantics to WFPT at the cost of an extra object replica.

8. CONCLUSION AND FUTURE WORK

In this paper we introduced the design of a new, predictable communication mechanism for real-time systems called wait-free pair transactions. WFPTs provide consistency without requiring blocking and have constant time overheads with bounded spatial constraints. We described a prototype implementation of WFPTs in the context of Fiji VM. Our implementation supports WFPTs within a single VM as well as across VMs in a multi-VM deployment.

The optimizations described in Section 4.3 remain for future work. We anticipate greatly improved performance when accessing WFPTs with multiple fields under such optimizations. The multi-VM jPapaBench application, for example, uses WFPTs with ten or more fields used in combination, so WFPTs with multiple fields are a realistic use case. Other optimizations specific to inter-partition WFPTs are possible, as well, such as partial type system reconciliation between VMs to speed type casts and optimization of field accesses based on the VM's fixed reader/writer context with respect to a given WFPT.

Inter-partition WFPT objects as implemented in this work cannot contain array fields, including even arrays of primitives. This restriction could be relaxed under certain circumstances, with primitive arrays allocated out of the same immortal space as the WFPT object and array reference assignments protected by a similar mechanism to SCJ or RTSJ scope checks.

Acknowledgments

We would like to thank Filip Pizlo for his work and input during the development of the wait-free pair transaction model. Portions of the WFPT implementation published here were produced under NSF SBIR Award #1248583. All findings and opinions are those of the authors, and are not supported or endorsed by the NSF.

9. REFERENCES

[1] Paparazzi project. `http://paparazzi.enac.fr/wiki/Main_Page`.

[2] Aicas. *JamaicaVM*. `http://www.aicas.com/jamaica.html`, 2010.

[3] Aonix. *PERC Products*. `http://www.aoinix.com/perc.html`, 2010.

[4] Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A real-time Java virtual machine with applications in avionics. *ACM Transactions in Embedded Computing Systems*, 7(1):1–49, 2007.

[5] Albert M. K. Cheng and James Ras. The implementation of the priority ceiling protocol in Ada-2005. *Ada Letters*, XXVII(1):24–39, 2007.

[6] J. B. Goodenough and L. Sha. The priority ceiling protocol: A method for minimizing the blocking of high priority Ada tasks. In *Proceedings of the Workshop on Real-time Ada Issues*, IRTAW '87, pages 20–31. ACM, 1988.

[7] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[8] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proceedings of the Symposium on Principles and Practice of Parallel Programming*, PPoP '05, pages 48–60. ACM, 2005.

[9] Tomas Kalibera, Pavel Parizek, Michal Malohlava, and Martin Schoeberl. Exhaustive testing of safety critical Java. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 164–174, 2010.

[10] Jeremy Manson, Jason Baker, Antonio Cunei, Suresh Jagannathan, Marek Prochazka, Bin Xin, and Jan Vitek. Preemptible atomic regions for real-time Java. In *Proceedings of the Real-Time Systems Symposium*, RTSS '05, pages 62–71, 2005.

[11] Vincenzo Nicosia. Towards hard real-time Erlang. In *Proceedings of the SIGPLAN Workshop on ERLANG*, ERLANG '07, pages 29–36. ACM, 2007.

[12] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '10, pages 146–159, 2010.

[13] Filip Pizlo, Lukasz Ziarek, and Jan Vitek. Real time Java on resource-constrained platforms with Fiji VM. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 110–119, 2009.

[14] John H. Reppy. *Concurrent programming in ML*. Cambridge University Press, 1999.

[15] Martin Schoeberl, Hans Sondergaard, Bent Thomsen, and Anders P. Ravn. A profile for safety critical Java. In *Proceedings of the Symposium on Object and Component-Oriented Real-Time Distributed Computing*, ISORC '07, pages 94–101, 2007.

[16] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computing*, 39(9):1175–1185, 1990.

[17] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, 1995.

[18] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in stm. *SIGPLAN Not.*, 42(6):78–88, 2007.

[19] H. R. Simpson. Four-slot fully asynchronous communication mechanism. *IEE Proceedings*, 137(1):17–30, January 1990.

[20] Adam Welc, Antony L. Hosking, and Suresh Jagannathan. Preemption-based avoidance of priority inversion for Java. In *Proceedings of the Conference on Parallel Processing*, ICPP '04, pages 529–538, 2004.

[21] Lukasz Ziarek. PRP: priority rollback protocol — a PIP extension for mixed criticality systems: short paper. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '10, pages 82–84, 2010.

[22] Lukasz Ziarek, KC Sivaramakrishnan, and Suresh Jagannathan. Composable asynchronous events. In *Proceedings of the Conference on Programming Language Design and Implementation*, PLDI '11, pages 628–639, 2011.

[23] Lukasz Ziarek, Adam Welc, Ali-Reza Adl-Tabatabai, Vijay Menon, Tatiana Shpeisman, and Suresh Jagannathan. A uniform transactional execution environment for Java. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP '08, 2008.