

Asynchronous Exception Propagation in Blocked Tasks

Roy Krischer and Peter A. Buhr
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{rkrische,pabuhr}@uwaterloo.ca

ABSTRACT

Asynchronous exception propagation is a useful alternative form of communication among threads, especially if timely propagation is ensured. However, timely propagation is impossible for blocked threads. An approach is presented to transparently unblock threads to begin propagation of asynchronous termination and resumption exceptions. The approach does not require additional syntax, simplifies certain programming situations, and can improve performance.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features—*concurrent programming structures, control structures*

General Terms

Exception Handling, Asynchronous Exceptions, Resumption

1. INTRODUCTION

A sequential program has one *execution* (stack), so an exception is propagated and handled inside its originating execution. A complex sequential program (*e.g.*, employing coroutines or continuations) and a concurrent program have multiple executions often interacting with each other; thus, an exception in one execution can be relevant to another, or exceptions can be used to communicate and facilitate a transfer of control in another execution. Therefore, mechanisms are necessary for exceptions to cross execution boundaries and executions to react to these exceptions.

1.1 Definitions

Unlike a regular *synchronous* raise (*e.g.*, `throw`), which flows into propagation directly, an *asynchronous raise* separates the execution paths of raise and propagation (see also [3]). The *raising execution* performs an asynchronous raise at the *propagating execution*. In particular, the propagating execution may not need to perform any special operations to receive the exception, and propagation can start at arbitrary

execution points. The result is non-determinism in the propagating execution not present with a synchronous raise. In detail, asynchronous exception handling involves:

1. *Raise*: a raising execution executes an asynchronous raise statement (analogous to synchronous raise).
2. *Delivery*: responsibility for the exception is transferred to the propagating execution.
3. *Detection*: a delivered exception is examined by the propagating execution and, if eligible, is propagated.
4. *Propagation*: involves handler matching, possible stack unwinding, etc., in the propagating execution.
5. *Handling*: the exception is matched and control transfers to a designated handler.
6. *Return*: when a handler completes (no new raise), control transfers to a point after the handler (*termination*) or detection point (*resumption*).

Propagation, handling, and return are identical to the synchronous case, while detection and delivery are unique to asynchronous raise. Detection can occur in the propagating execution under (*full*) *asynchrony*, *i.e.*, at arbitrary points, or under *restricted asynchrony*, *i.e.*, at well defined points, to mitigate some non-determinism. Implementations may combine some steps, *e.g.*, raise and delivery, and asynchronous raise can originate in the run-time system.

1.2 Motivation

Exceptional situations may be urgent (emergencies), requiring immediate propagation and handling. For synchronous raise, this requirement is met by the immediate start of exception propagation at the raise. For asynchronous raise, this requirement may not be met when an exception is raised at a blocked execution. If propagation is delayed until the execution unblocks, urgency is forfeit. Therefore, algorithms assuming that asynchronous exceptions are propagated immediately face a potential unbounded wait. For example, imagine a number of tasks synchronizing on a barrier (*e.g.*, at the end of an atomic action). If one task fails and exceptions are raised to inform the others, how can these tasks react if blocked on the barrier? While it may seem obvious these tasks should be unblocked, there are significant semantic and technical issues in allowing timely unblocking to occur.

This paper proposes that the delivery of an exception to a blocked task should unblock it so propagation can begin immediately. This semantics follows directly from the no-

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

WEH '08, November 14, Atlanta, Georgia, USA
Copyright 2008 ACM 978-1-60558-229-0 ... \$5.00.

tion of termination: propagation aborts the current operation, so waiting for this operation should be aborted, too. The contribution of this work is the development of a comprehensive design for exceptional unblocking semantics, a prototype implementation, and examples with performance results demonstrating the usefulness of this feature. The language employed is $\mu\text{C++}$, a concurrent dialect of C++ (see [2] for details), whose exception handling facilities are modeled after [6]. $\mu\text{C++}$ exceptions are explicitly raised at propagating executions with detection following restricted asynchrony implemented through polling. Both termination and resumption semantics are supported, as well as bound-object matching [5]. $\mu\text{C++}$ is well suited for demonstrating different design and implementation issues because it supports both termination and resumption, as well as a variety of blocking instruments of varying complexity.

1.3 Related Work

A number of publications address extending exception semantics to a multi-execution domain, *e.g.*, [14, 8, 10, 9, 12, 6, 11], and a few main-stream languages support some form of multi-execution exception handling. Ada [15] allows an exception to cross execution boundaries during rendezvous synchronization. Java’s Thread class supports a stop method, which facilitates thread cancellation but can be used to raise arbitrary Throwable objects inside the called thread. An exception raised in this way between two executions (threads) is fully asynchronous. However, it is not permitted to catch an exception raised through the stop mechanism unless it is reraised; thus, the mechanism is only suitable for thread cancellation as opposed to general asynchronous exception handling. Furthermore, Thread.stop is deprecated in current Java versions for safety reasons [1]. Java supports thread *interruption* through Thread.interrupt, which can interrupt a thread blocked on certain calls (restricted asynchrony) but cannot directly raise an arbitrary exception. POSIX threads [7] also support thread cancellation, which can either be (fully) *asynchronous*, or *deferred* (restricted asynchrony) with checking at cancellation points in a limited number of system routines. A cancelled task blocked on a cancellation point is unblocked and cancellation begins immediately. Note, thread cancellation does not constitute exception handling and only supports a very limited form of termination. Java and POSIX do not support resumption or certain high-level concurrency concepts, *e.g.*, rendezvous.

2. DESIGN CONSIDERATIONS

Since a blocking operation usually involves a routine call, intuitively, this call should be perceived as responsible for raising the exception. Whether a potentially blocking call succeeds immediately or after some blocking is usually transparent to the programmer. Analogously, the perceived control flow should not differ between the case of an exception that propagates immediately upon calling the blocking routine and that of the exception propagating after the thread has been blocked for some time. Neither should the raising propagation have to concern itself with whether the propagating execution is blocked. In this way, the program behaves consistently in both cases with predictable control flow. Given the identical control flow of the blocked and non-blocked exceptional propagation, it would be useful if the local state, *i.e.*, the state of the propagating task and data it accesses

in connection with the blocking operation, could be identical. The point in time immediately preceding a blocking call and in which an exception can potentially be detected (*e.g.*, through polling) shall be denoted t^- . Similarly, t^+ is the point in time immediately succeeding a blocking call when a task becomes active (after being blocked) and an exception can be detected (*e.g.*, through polling). Time t is between t^- and t^+ , when a task is blocked and a pending exception is detected. Rephrasing the above design goal formally, the control flow resulting from an exception detected at t shall appear to be identical to that resulting from an exception detected at t^- . The following is an analysis and description of detailed semantics for the different scenarios in which a task can block. While the studied language, $\mu\text{C++}$, determines some details of the discussion, the overall analysis can apply equally to any language with similar blocking instruments. Terminating semantics (**throw**) are analyzed first; resumption semantics are added to the design, subsequently.

2.1 Mutex Lock

The simplest blocking scenario is a mutex (owner) lock, *e.g.*:

```
mutexLock lock;
try {
    asyncPoll();           // poll detects exceptions ( $t^-$ )
    RAllacquire x(lock);   // acquire lock ( $t$ )
    asyncPoll();           // poll detects exceptions ( $t^+$ )
} catch (...) {...} // handler
```

Routine `asyncPoll` checks for pending asynchronous exceptions delivered to the execution (task). The explicit poll checks for pending exceptions (at t^-) before the potentially blocking call to `acquire`, but an implicit detection before the call is also possible. In both cases, a terminating propagation transfers control to the handler. To satisfy the proposed semantics, the behaviour upon detection of an exception when blocked on `RAllacquire` (at t) should appear identical to the previous cases, *i.e.*, after detecting the exception, the task is unblocked and propagation begins out of the routine call. Note, by using the ‘resource allocation is initialization’ technique [13] in this case, acquiring the lock is exception-neutral, *i.e.*, the lock acquisition is undone (released) after exception propagation without hindering the propagation. Therefore, an explicit (or implicit) poll just after the call (at t^+) resulting in propagation has the same control flow. This consistency across the three different detection points argues in favour of the proposed design. Similar mutex/synchronization instruments can be treated analogously. For example, if an exception is delivered asynchronously to a task waiting in the P routine of a semaphore, the task is unblocked, P acts as the source of the exception, and the semaphore counter is adjusted to account for the unblocking task.

2.2 Monitor

A monitor provides mutual exclusion at entry as well as task synchronization by waiting (blocking) on a condition variable or accepting from a mutex member. Figure 1 shows the implementation structure for the mutual exclusion and synchronization of a $\mu\text{C++}$ monitor. A `wait` on a condition variable blocks the monitor owner on the specified condition-variable waiting-queue. A `signal` on a condition variable moves the task at the head of the condition queue (*signallee*)

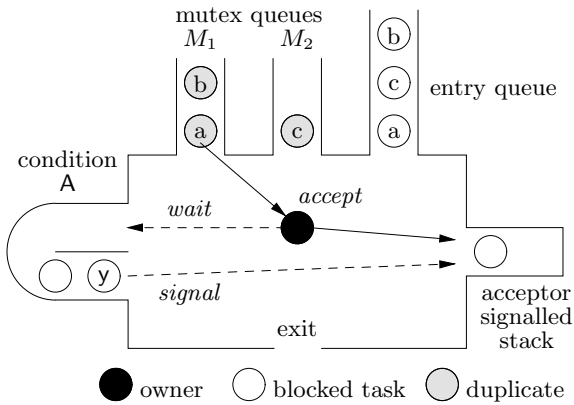


Figure 1: $\mu\text{C++}$ monitor

to the signalled stack and the monitor owner (*signaller*) continues. $\mu\text{C++}$ also supports an `_Accept` statement to explicitly schedule tasks using rendezvous from outside the monitor. An `accept`, e.g., `_Accept M1`, blocks the owner task on the acceptor stack and the task at the head of the specified mutex (member routine) queue becomes the owner, similar to Ada’s `select/accept` for tasks.¹ In general, the scheduling order is: when the monitor owner exits or waits, the head of the acceptor/signalled-stack (the most recent task signalled or whose accepted member is called) is scheduled (LIFO order). If there is no such task, the head of the entry queue gains ownership (FIFO order, subject to real-time priorities). See [2, §2.9] for more details.

2.2.1 Entry

A call into a monitor is similar to lock acquisition, so the same semantics can be used to achieve the desired properties, i.e., unblock the task and let the propagation appear as if it originates from the call into the monitor. Again, consistency of control flow between t^- and t is obvious. At t^+ , immediately after entering the monitor, an entering task owns the monitor, but an exception propagation causes it to release ownership and to continue with control flow just like in t^- and t , which is the same as the mutex lock case.

2.2.2 Condition Variable Synchronization

When a task issues a `wait` on a condition variable at t^- , i.e., before it blocks, the task owns the monitor. Hence, if an exception is detected at t^- , it is propagated and handled while the task owns the monitor. For an exception detected at t , i.e., after the `wait`, another task may have been scheduled in the monitor and possibly made state changes. If exception propagation to a blocked task requires changing state inside the monitor, the propagating task must re-acquire the monitor, which delays exception handling. It would be preferable if the exception could be propagated directly from the routine call through which the propagating task entered the monitor, bypassing normal stack unwinding. In this way, there would be no need to compete for monitor ownership as propagation would take place outside of it. However, such a solution is infeasible. First, due to the bypassing, it violates the requirement for the control flows at t^- and t to be identical. Second, the stack needs to be unwound prop-

¹In $\mu\text{C++}$, the `accept` concept is generalized across any kind of mutex object, e.g., coroutine, monitor, or task.

erly between the call to `wait` and the entering call, which may require running cleanups that assume monitor ownership to maintain invariants. The same cleanup considerations also apply to handlers located inside the monitor, and it would be unnatural to ignore such handlers in the event of an asynchronous exception while blocked. Hence, the sensible design is to unblock the task, gain monitor ownership, which may require additional waiting on the signalled-stack or some special queue, and have the exception propagate from the call to `wait`. The advantage of this design is that control flow at t^- and t appear identical. The disadvantages are that local state and monitor ownership can change between t^- and t , which can invalidate the advantage of identical control flow. Furthermore, with the potential need to compete for monitor ownership, there is no guarantee for timely handling of the exception. In fact, depending on scheduling performed after t , the propagating task can be delayed indefinitely. Finally, by raising an exception at a task blocked on a condition variable, the raising execution implicitly influences the scheduling inside a monitor of which it possibly knows nothing. If the task is blocked on the signalled-stack at t (waiting to re-acquire the monitor) or the detection of an asynchronous exception causes it to be moved to some other queue, the exception handling process cannot be accelerated much further. The task must wait until the monitor becomes available before propagation can start when it is scheduled next. This restriction means the earliest the propagating task can execute is the time when the owner of the monitor (at t) relinquishes its ownership.

2.2.3 Accept Synchronization

With `accept` synchronization, the analysis for a call by an acceptee can be treated like an entry call. The analysis for the acceptor task is more complex, with separate cases at t for before and after an acceptee enters the monitor.

In the first case, i.e., if the exception is detected before a rendezvous begins, e.g.:

```
try { // try-block guarding entire _Accept statement
    _Accept ( mem1, mem3 ) { ... }
    or _Accept ( mem2 ) { ... }
} catch ( Ex ) { ... } // handler
```

the general semantics remain the same, i.e., the exception detection causes the acceptor to unblock, terminating the rendezvous similarly to a rendezvous time-out [4]. Then exception propagation begins as if originating from the `_Accept` statement and is caught by the handler, which may have to undo the *acceptor’s* actions because the rendezvous did not occur. Furthermore, the acceptor task had ownership of the monitor before it `accept`-blocked, and since no task has performed a monitor call before the exception, the propagating task can re-acquire ownership immediately. Hence, no local state change between t^- and t is possible, so consistency between propagations at t^- and t is maintained.

In the second case, i.e., an acceptee is executing a monitor call at t , so the acceptor task cannot be unblocked from the acceptor stack until it can obtain ownership of the monitor. The same analysis as in the condition variable case applies as the acceptor task is blocked on the acceptor stack; specifically, propagation starts as soon as the propagating task is rescheduled inside the monitor, similar to the behaviour at

t^+ . However, since a monitor call was accepted, there exists a responsible `_Accept` clause matching the rendezvous. This additional information is necessary if the acceptor detects an asynchronous exception and needs to undo the *acceptee's* actions. To support this case, the `accept` statement is extended with a `try`-block specific to an `_Accept` clause:²

```
_Accept ( mem1, mem3 ) try { ... } // specific try
  catch ( Ex ) { ... }           // and handler(s)
or _Accept ( mem2 ) try { ... }  // specific try
  catch ( Ex ) { ... }           // and handler(s)
```

where propagation of the asynchronous exception starts *inside* the `try`-block, *i.e.*, no propagation can take place between the `_Accept` clause and the `try`-block. Apart from this property, this `try`-block is the same as an enclosing one, and it guards against exceptions detected while blocked (at t) as well as those detected while executing code inside it (after t^+). The control flow at t is now the same as the control flow at t^+ as both see the exception arriving within an `accept's` `try`-block. It might be argued this behaviour now violates matching control flow at t^- and t . But unlike the previous cases, there is a fundamental difference here between t^- and t .³ At t^- , the rendezvous has not begun yet, whereas at t the rendezvous is already underway and the acceptor task is blocked merely due to internal scheduling. If the programmer chooses not to take advantage of the specific `try`-blocks and instead guards the entire `_Accept` statement, the control flows at t^- , t , and t^+ appear identical. However, since the *acceptee* was allowed to execute within the monitor, a local state change could have occurred after t^- .

2.2.4 Scheduling Considerations

The desire to wake blocked tasks to handle exceptions is based on the goal of quick exception handling. The minimum action the run-time system can take is to make the propagating task schedulable, thus, aborting the blocking operation. This minimal approach is based on the philosophy that the exception is a phenomenon localized to the propagating task and should not affect the rest of the system. Alternatively, it is conceivable to think of an exception as the manifestation of a situation affecting the entire system, and the propagating task is merely designated to deal with it. In this view, a propagating task should probably have precedence over non-propagating tasks in scheduling decisions as it is responsible for rectifying a potentially threatening state for the entire system. These two alternative philosophies also affect other design decisions, *e.g.*, if an exception cannot be handled, should just the propagating task be terminated or the entire program?

To be clear, the question of preferred treatment of propagating tasks only exists for equal-priority tasks. Regardless of the philosophy regarding local/global effects of exception handling, the explicit prioritization of tasks is a more fundamental method of determining task precedence. Thus, a ready higher-priority task should never be disadvantaged by a propagating lower-priority task. Otherwise, analyses using priorities cannot be applied any more (hard/soft real-time). However, an asynchronous exception is a communication be-

tween tasks, and it can be said that the propagating task handles the exception on behalf of the raising task. Thus, an argument can be made that the propagating task should temporarily inherit the effective priority of the raising task at the time of the raise if that priority is higher than its own.

When tasks have the same priority, there are three basic strategies for incorporating propagating tasks into internal monitor scheduling: promoting, demoting, and neutral. The promoting strategy gives propagating tasks precedence over normal tasks, *e.g.*, by placing them in a special list scheduled before any other. Conversely, the demoting strategy schedules any other task before a propagating one, *e.g.*, by placing a propagating task at the end of the signalled-stack. The neutral strategy treats propagating and normal tasks alike, in principle. Still, there are various nuances to implement such a strategy, and thus, bias the scheduling on a subtler level. The simplest neutral strategy is to treat the propagating task as if it had been signalled at the moment of detection, *i.e.*, it is moved to the top of the signalled stack. This approach is equivalent to signalling the propagating task, which subsequently polls for exceptions after waking up. However, this *exceptional signal* is sent by the raising task, which may, unlike a regular signal in $\mu C++$, not be the owner of the monitor.⁴

In the demoting strategy, the time until a propagating task can execute is *at least* that of a normal signal with subsequent poll by the waking task, but likely more due to other tasks scheduled ahead. With a neutral strategy, other tasks can be scheduled ahead of the propagating tasks, which may result in no speedup of exceptional propagation compared to a normal signal and poll. With a promoting strategy, propagating tasks are scheduled at the earliest possible time, *i.e.*, when the current monitor owner relinquishes ownership. However, this scheduling makes it more difficult to reason about the order of execution after successful synchronization since asynchronously arriving exceptions perturb the normal scheduling order. Nevertheless, this effect cannot be avoided if the goal of the promoting strategy is to favour propagating tasks over the normal ordering for timely execution. Note, scheduling perturbation is only noticeable in monitors with well-defined scheduling order; monitors with no strict ordering, *e.g.*, as in Java, have nothing to perturb.

Choosing the right strategy depends on which compromise is preferred between predictable scheduling order and quick exception handling. Using a demoting strategy, the normal scheduling order is preserved, but the handling of the exception can be delayed significantly. Conversely, a promoting strategy sacrifices predictability in favour of quick exception handling. Neutral strategies are the least useful because neither scheduling order nor speedy exception handling are ensured. If the philosophy is to sacrifice scheduling order to expedite exception handling, the following assumption is helpful. The propagating task, aware of its potential interference with normal scheduling order, should not manipulate the monitor beyond necessary cleanups (including maintaining monitor invariants) and leave quickly. Hence, the actual time in which a propagating task interferes with synchronization and the extent of this interference, *i.e.*, manipula-

²This construct is similar to a C++ constructor `try`-block.

³This difference also exists for signalled tasks; however, there is no resulting difference in control flow.

⁴Note, there are systems that allow signalling without ownership, *i.e.*, from outside a monitor, *e.g.*, POSIX threads.

tion of shared data, should be minimal. For these reasons, a promoting scheduling strategy is employed in $\mu\text{C++}$.

3. RESUMPTION SEMANTICS

For resumption, *i.e.*, return to the conceptual raise (point of detection) after the handler completes, the main goal is to ensure consistency of control flow between t^- and t . Furthermore, resumption must be consistent with the semantics discussed in the previous sections dealing with terminating exceptions, which is particularly important if a resumption handler chooses to (re-)raise an exception. For example, if an exception is detected at t^- , the resumption handler is executed before a blocking call:

```
try {
    // resumption exception at t^-
    // blocking call
} resume( Ex ) { ...if (...) throw; ... } // resumption handler
} catch( Ex ) { ... } // termination handler
```

Ultimately, the resumption handler must exit either by a (re-)throw or a return. If the handler (re-)throws, propagation unwinds all handler frames until it reaches the point of the resumption detection, from which point control flow is indistinguishable from a termination exception detected at t^- . If the handler returns, the task proceeds by issuing the potentially blocking call. Similar behaviour is required at t . The task must unblock and execute the resumption handler. However, unlike with termination semantics, the task is still conceptually attached to the mutex/synchronization instrument and must therefore remain *pseudo-blocked* on it. Hence, even though the task is scheduled for execution, any lock accounting information, such as a semaphore counter, is maintained as if the task is still blocked. If the handler (re-)throws, the behaviour past the detection point shall be identical to terminating semantics. If the handler returns, resumption semantics require the task to proceed from the point of detection, which can be achieved by returning to the blocked state at the location where it blocked originally. The safety of this *reblocking* in general relies on the fact that the original call did not proceed beyond t (since the resumption handlers' call sequence is a fork off the task's regular execution path). This semantics requires the reblocking task to evaluate its state as well as that of the blocking instrument, possibly foregoing the reblocking, *e.g.*, if the task has been signalled while executing the resumption handler.

This example of waiting on a condition variable shows why re-issuing the blocking call without recognizing the retry does not suffice. It also demonstrates why a resumption should not unblock a task fully but instead keep it pseudo-blocked. Since signals are not stored in a condition variable, *i.e.*, there is no counter, a common idiom is for the signalling task to set a flag in the monitor so a waiting task can determine whether it should wait:

```
_Monitor Mo {
    bool flag;
    uCondition cv;
public:
    Mo() : flag(false) {}
    int maybeWait() { if ( !flag ) cv.wait(); } // task A
    void wakeup() { flag = true; cv.signal(); } // task B
};
```

Assume task A calls `maybeWait` and waits on condition `cv` because the flag is not set. Suppose task B now calls `wakeup`, acquires ownership, and immediately thereafter, a pending resumption (raised from outside the monitor) for task A is detected. As a result, A is unqueued from `cv` and put on the signalled stack due to promoting scheduling (see Section 2.2.4). Task B continues inside `wakeup`, sets the flag, signals `cv`, which is empty so the signal is lost, and leaves. Propagating task A is now scheduled and executes its handler (omitted above), and, upon return, rewaits on `cv`. If this rewait just blocks without rechecking the flag, task A does not realize its signal has occurred but been lost. While there are explicit programming approaches to solve this problem, it is easiest to have the condition variable implicitly manage the propagating task while it is pseudo-blocked. Hence, when the propagating task tries to rewait, the signal is attributed to it, and it is unqueued instead.

4. IMPLEMENTATION

Several challenges need to be addressed in the implementation of asynchronous exception handling by blocked tasks.

4.1 General Steps

The first challenge is providing for asynchronous delivery and detection. The former is achieved by chaining the exception to an asynchronous exception-queue associated with the propagating task, and the latter by inserting implicit poll points into the code before blocking and after unblocking. Inserting a poll point before a blocking operation (at t^-) makes sense since it is illogical for a task to block with exceptions in its queue that would cause it to become unblocked (at t). Inserting after a task's unblocking is necessary to force propagation of the exception before further advance.

The second challenge is to determine whether the propagating task is blocked. This check and any ensuing actions, *e.g.*, detecting the deliverability of the exception, unblocking the propagating task, etc, must be performed by some active task. The *delivering task* (in $\mu\text{C++}$, the raising task) is an obvious choice for performing these actions because it is active and already has to manipulate the propagating task as part of exception delivery. The information about a propagating task's running state is maintained in the task itself and can be checked easily, but the difficulty is avoiding the race condition inherent in this check. One possible solution is to have a delivery lock broadly guarding the blocked-check and all subsequent operations by the delivering task as well as the poll and block/unblock by the propagating task. This method ensures that once an asynchronous exception is being delivered, the propagating task cannot block/unblock (except on the delivery lock). Its disadvantage is the potential loss of concurrency, *e.g.*, the propagating task unblocks from the original blocking operation and could process its exception queue, but instead blocks on the delivery lock because a delivery is occurring. An alternative approach is for the delivering task to acquire the delivery lock just before it causes the propagating task to unblock. This method complicates the implementation significantly because it needs to ensure the deliverability of the asynchronous exception can be determined by the delivering task despite the propagating task's concurrent execution. Ensuring this safety alone can invalidate any performance advantage. Furthermore, the delivering task must verify whether its exception has already

been propagated by an awakened propagating task. The original broadly-locking approach avoids these issues and is therefore preferable.

The third challenge is how to activate a blocked task in order for it to process its exception queue. If the task is spinning on a spin lock, it can check its own exception queue or some flag that is set by the delivering task and no further action is required by the delivering task. For blocking instruments, the delivery task needs to actively perform some administrative action in order to unblock the propagating task, *e.g.*, moving the propagating task off some waiting queue and onto a ready queue. The propagating task should provide a method or at least additional information that the delivering task can use to unblock it (since the propagating task knows on what instrument it is blocked at that moment). Most likely, the delivering task has to acquire some lock protecting the internal data structures of the blocking instrument, which may force it to block, but these locks are usually designed to be acquired for a short time only. In general, the waking and unblocking of a task due to an exception is similar to a time-out; hence, if time-out facilities exist, they could be exploited for the exception case. The following is a description of the detailed $\mu\text{C++}$ implementations with regards to the respective blocking instruments.

4.2 Mutex Lock / Monitor Entry

For any blocking lock, it suffices to acquire its (internal) protecting lock and make the blocked task ready. As the propagating task polls after waking, it must detect the pending exception and begin propagation instead of entering the critical region. Simple monitor entry is implemented similarly.

4.3 Monitor Condition Variables

For tasks waiting on a monitor condition variable, the implementation is more complex since the propagating task needs to compete for monitor ownership. Monitor semantics vary, resulting in different implementations, and these implementation details determine how the propagating task is awakened. This implementation assumes the $\mu\text{C++}$ monitor semantics, where only the monitor owner can access the various internal waiting queues/stack. To facilitate the scheduling of propagating tasks, the current monitor owner needs to be made aware of their presence. If no monitor owner exists, the delivering task itself must enter the monitor and perform the necessary actions. To avoid a race condition for detecting whether or not a monitor owner exists, the leaving/waiting owner has to perform additional locking. The required action, *e.g.*, moving a propagating task to the signalled stack, needs to be encoded in an *action queue* the monitor owner processes when relinquishing ownership. The delivering task executes the following protocol:

- acquire delivery lock, queue exception, check if propagating task is blocked,
- if not blocked, release delivery lock, done.
 - (otherwise) verify exception’s eligibility for propagation,
 - if exception is disabled, release delivery lock, done.
 - * (otherwise) acquire monitor lock, add action to action queue, check for a monitor owner
 - * if there is owner, release locks, done.
 - (otherwise) execute actions, release locks, done.

Finally, the propagating task needs to poll for exceptions as soon as it wakes up, processing one termination or all resumption exceptions.

Since the delivering task cannot manipulate the internal monitor queues directly (unless it owns the monitor), and it would be inefficient for the delivering task to wait until it can own the monitor, it needs to communicate the desired scheduling to the monitor owner. The action queue is an efficient mechanism for this communication since it needs to be processed only once and only upon relinquishing ownership of the monitor, which is the time at which the monitor owner makes scheduling decisions in any case. The disadvantage is that it complicates the implementation of certain neutral scheduling strategies. For example, consider the neutral strategy in which an asynchronous exception delivery is interpreted as an exceptional signal, requiring the propagating task to logically occupy the top of the signalled stack at that moment. However, the raising task is not the owner, so this requirement is recorded in the action queue. Then, the current monitor owner signals condition variables, pushing tasks onto the signalled stack. So when ownership is relinquished and the action queue is processed, precise temporal information about when the exception was detected with respect to the signalled tasks is unavailable (or needs to be recorded/recovered); hence, replicating the scheduling order required by this neutral strategy is difficult. However, neutral strategies produce the least useful scheduling (see Section 2.2.4). As well, this strategy tries to enforce an ordering that is, due to the asynchronous nature of the exception, inherently non-deterministic. Hence, no advantage can be gained by following this strategy, and thus, precluding its use due to the action-queue implementation is acceptable.

4.4 Accepting

Rendezvous using `_Accept` needs to be implemented similarly, with the following additional considerations. If the rendezvous has not occurred, the acceptor can simply be unblocked. If a rendezvous has occurred, the acceptor must be on the acceptor stack and no further action is required.

4.5 Resumption

Supporting resumption adds more implementation complexity. Unlike termination, which occurs once and always aborts the blocking operation, multiple different resumptions can occur while a task is blocked, resulting in multiple transitions between running pseudo-blocked and reblocking (one for each resumption exception handled). Hence, polling and subsequent reblocking may repeat when a propagating task is awakened via a resumption. Pseudo-blocking can cause further complications, *e.g.*, when an acceptee arrives while the acceptor is pseudo-blocked. Resumption can also allow a task to re-enter a monitor, and this task must only acquire monitor ownership once along the entire pseudo-blocked re-entry chain. Furthermore, a task can block again on a condition variable on which it is still pseudo-blocked, or accept a member while pseudo-blocked on an `_Accept` statement (or any arbitrarily complex combination/repetition of these situations). While such program logic does not seem advisable, it cannot be rejected, and thus, needs to be addressed by the implementation. Additional information indicating whether tasks are blocked normally or pseudo-blocked is therefore required. Then, a propagating task’s transitioning

from blocked to pseudo-blocked merely requires it to be designated schedulable, and, if required, added to the signalled-stack according to the monitor’s scheduling strategy. If a pseudo-blocked task is unblocked normally (*e.g.*, its condition variable is signalled, or it obtains a lock or monitor), it needs to be unqueued just like a regularly-blocked task, so after returning from the handler, the reblocking operation simply returns and the task proceeds. Otherwise, if the task has not been signalled before its handler completes, it needs to reblock and the pseudo-blocked flag is removed. Distinguishing between a new blocking operation (*e.g.*, entry, wait, `_Accept`) and a reblocking can be achieved by associating a stack-allocated object containing the pseudo-blocked flag for each unique blocking operation by a task.

5. EMPIRICAL RESULTS

Two aspects of the new language feature are important: what are the effects in terms of power of expression and ease of use, and what are the effects on run-time performance? These aspects are evaluated through example programs. (Program syntax is idealized for simplicity; actual programs use more complex $\mu\text{C++}$ syntax.)

5.1 Worry-Free Synchronization

Figure 2 shows a server task asynchronously providing a computationally expensive service to a number of clients. Two versions of the program are present, with and without the new unblocking semantics. Client and server follow a simple protocol: a client starts a computation by calling `Server::sendRequest`, which retains the client id and request for asynchronous processing through the server while the client does other work; a client calls `Server::getResult` to obtain the result. Note, server calls are synchronous, and hence, may block the client. Assume some client inputs are faulty, which the server detects in half the time it takes it to perform a computation, aborting the computation. The server’s catch clause handles the faulty case (`CompError`) by relaying the exception asynchronously to the responsible client, which may be working or waiting for the result. Without the new semantics, the client cannot respond to the exception if it is blocked waiting for the result. Hence, it is necessary for the server to complete the synchronization protocol by accepting `Server::getResult` in the handler (`NoUnblocking` version), even though there is no result, so the client can unblock and propagate the exception. With the new semantics, this additional call is unnecessary as the exception delivery wakes the client. As well, without the new semantics, it is necessary for the client to poll at the start of `Server::getResult` and subsequently receive the `CompError` exception in order to ensure it does not return an arbitrary result and proceeds to use it. Such an addition is unnecessary with the new semantics. To summarize, as soon as asynchronous exceptions influence control flow, synchronization becomes more complicated without the new semantics as special precautions need to be taken when a propagating task is blocked. With the new semantics, no extra code is required as the raise automatically does the right thing. As a side effect, the program also becomes more efficient because blocking due to additional synchronization is avoided: without the new semantics, the runtime is 12.00s; with them, the runtime is between 9.40s and 9.41s (10 runs each).

5.2 Why not Cheat and Run While Blocked?

Figure 3 consists of a number of `Worker` tasks, each operating on a distinct portion of data. To calculate a result, a worker does `prework` independently and then work inside a common monitor. However, some of the values supplied to the workers are erroneous. The main task therefore sends out messages (as resumptions) to revoke the faulty values and trigger a (re-)calculation. Note, this situation is a natural application for asynchronous resumptions as the correcting action is an independent fork off (and return to) the main control path of the worker tasks. In this example, the same code is used for testing old and new semantics; however, two explicit poll points have been injected to increase performance with the old semantics. Nevertheless, using the new semantics yields a run-time of 10.95s compared to 15.26 ± 1 s without it (ten runs each). This difference can be explained by pseudo-blocking. With the new semantics,

```

#define ms * 1000000
_Event CompError {};
_Task Server {
    uBaseTask *c; int run, result, req;
    int compute( int ) throw ( CompError ) {
        _Timeout( uDuration( 0, 50 ms ) ); // work
        if ( ++run % 3 == 0 ) _Throw CompError();
        _Timeout( uDuration( 0, 50 ms ) ); // work
    }
public:
    Server() : run( 0 ) {}
    void sendRequest(int n) {c = &uThisTask(); req = n;}
    int getResult() {
        asyncPoll(); // NoUnblocking
        return result;
    }
    void main() {
        for ( ;; )
            try {
                _Accept( ~Server ) { break; }
                or _Accept( sendRequest ) {
                    result = compute(req);
                    _Accept(getResult);
                }
            } catch( CompError ) {
                _Throw _At *c;
                _Accept( getResult ); // NoUnblocking
            } // try
    } // main
} server;
_Task Client {
public:
    void main() {
        for ( int i = 0 ; i < 20 ; i += 1 ) {
            server.sendRequest( i );
            try {
                _Timeout( uDuration( 0, 150 ms ) ); // work
                int res = server.getResult();
            } catch( CompError ) {}
        } // for
    } // main
};
void uMain::main() {
    uProcessor p[2]; // create kernel threads
    Client c[4]; // create client tasks
}

```

Figure 2: Client/Server

tasks that are lined up to enter the monitor and receive a resumption can step out and at least complete `prework` while still being conceptually blocked on monitor entry (and without losing their spot in the queue). Without the new facility, a task cannot react to the resumption until it gains ownership of the highly-contested monitor. Indeed, the difference of 4.31 s is close to the theoretical maximum of 5 s (20×0.25 s of `prework`). Hence, the ability to run while conceptually blocked results in a substantial performance increase.

6. CONCLUSION

Allowing asynchronous exceptions to unblock their propagating task follows naturally from the wish to ensure timely handling of an exception. As demonstrated, such a feature can be implemented without the need for additional syn-

```
#define ms * 1000000
const int TASKS=8, CHUNK=10, SPACE=TASKS*CHUNK;
_Event Recall {
public:
    int i; int correction;
    Recall( int i, int c ) : i(i), correction(c) {}
};
_Monitor Mo {
public:
    void work(int &i) {_Timeout( uDuration( 0, 100 ms ) );}
} mo;
_Task Worker {
    int *data;
    void prework(int &d) {_Timeout(uDuration(0,250 ms));}
public:
    void start( int d[] ) { data = d; }
    void main() {
        bool done[CHUNK] = { false };
        _Accept( start );
        for ( int i = 0; i < CHUNK ; i += 1 ) {
            try {
                if ( done[i] ) continue; // iteration fixed ?
                prework( data[i] );
                asyncPoll();
                mo.work( data[i] );
                asyncPoll();
            } resume( Recall &r ) {
                data[r.i] = r.correction;
                prework( data[r.i] );
                mo.work( data[r.i] );
                done[i] = true;
                if ( i == r.i ) _Throw; // abort iteration
            } catch ( Recall ) { }
        } // for
    } // main
};
void uMain::main() {
    uProcessor p[TASKS]; // create kernel threads
    int array[SPACE];
    Worker w[TASKS]; // create tasks
    for ( int i = 0; i < TASKS; i += 1 ) // initialize/start
        w[i].start( &array[ i * CHUNK ] );
    _Timeout( uDuration( 1 ) );
    for ( int i = 0; i < SPACE; i += 4 ) { // fix every 4th
        _Resume Recall( i % CHUNK, 3 ) _At w[i/CHUNK];
        _Timeout( uDuration( 0, 100 ms ) );
    } // for
} // main
```

Figure 3: Correct Faulty Data with Resumptions

tax or unusual programming techniques. As well, especially with elaborate synchronization protocols, this language feature can allow the programmer to write simpler, more intuitive code. In addition, when there is strong contention for a shared resource, pseudo-blocking can be used to increase concurrency, and thus, program performance.

7. REFERENCES

- [1] *Java™ 2 Platform Standard Ed. 5.0 Documentation: java.lang.Thread*. <http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Thread.html>.
- [2] P. A. Buhr. $\mu\text{C}\text{++}$ annotated reference manual, version 5.5.0. Technical report, Sept. 2007. <http://plg.uwaterloo.ca/~usystem/pub/uSystem/uC++.pdf>.
- [3] P. A. Buhr, A. Harji, and W. Y. R. Mok. Exception handling. In M. V. Zelkowitz, editor, *Advances in COMPUTERS*, volume 56, pages 245–303. Academic Press, 2002.
- [4] P. A. Buhr, A. S. Harji, P. E. Lim, and J. Chen. Object-oriented real-time concurrency. *SIGPLAN Notices*, 35(10):29–46, Oct. 2000. OOPSL’00, Oct. 15–19, 2000, Minneapolis, Minnesota, U.S.A.
- [5] P. A. Buhr and R. Krischer. Bound exceptions in object-oriented programming. In *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag, 2006.
- [6] P. A. Buhr and W. Y. R. Mok. Advanced exception handling mechanisms. *IEEE Trans. Softw. Eng.*, 26(9):820–836, Sept. 2000.
- [7] D. R. Butenhof. *Programming with POSIX Threads*. Professional Computing. Addison-Wesley, 1997.
- [8] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Trans. Softw. Eng.*, 12(8):811–826, 1986.
- [9] Y. Ichisugi and A. Yonezawa. Exception handling and real time features in an object-oriented concurrent language. In *Proceedings of the UK/Japan workshop on Concurrency : theory, language, and architecture*, pages 92–109, New York, NY, USA, 1991. Springer-Verlag New York, Inc.
- [10] V. Issarny. An exception handling model for parallel programming and its verification. In *SIGSOFT ’91: Proceedings of the conference on Software for critical systems*, pages 92–100, New York, NY, USA, 1991. ACM Press.
- [11] M. Rintala. Handling multiple concurrent exceptions in $\text{C}\text{++}$ using futures. In *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 62–80. Springer-Verlag, 2006.
- [12] A. Romanovsky, J. Xu, and B. Randell. Exception handling in object-oriented real-time distributed systems, 1997.
- [13] B. Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.
- [14] A. Szalas and D. Szczepanska. Exception handling in parallel computations. *SIGPLAN Notices*, 20(10):95–104, Oct. 1985.
- [15] United States Department of Defense. *The Programming Language Ada: Reference Manual*, ANSI/MIL-STD-1815A-1983 edition, Feb. 1983. Published by Springer-Verlag.