Resilient Optimistic Termination Detection for the Async-Finish Model

SARA S. HAMOUDA and JOSH MILTHORPE,

Australian National University, Australia

[Technical Report]

Driven by the consistent trend of increasing core count and decreasing mean-time-to-failure in supercomputers, HPC runtime systems must improve support for dynamic task-based execution and resilience to failures.

The async-finish task model, adapted for distributed systems as the asynchronous partitioned global address space programming model, provides a simple way to decompose a computation into nested task groups, each managed by a 'finish' that signals the termination of all tasks within the group. A failure impacting one finish scope does not propagate to other finish scopes by default, providing an opportunity for local recovery, an important feature for resilient large-scale computations.

For distributed termination detection, maintaining a consistent view of task state across multiple unreliable processes requires additional book-keeping when creating or completing tasks and finish-scopes. Runtime systems which perform this book-keeping *pessimistically*, i.e. synchronously with task state changes, therefore add a high communication overhead compared to non-resilient protocols. Prior work presented scalable resilient protocols for Cilk-like spawn-sync task model, a subset of the more general async-finish model.

In this paper, we propose *optimistic finish*, a resilient termination detection protocol for async-finish. By avoiding communicating certain task and finish events, this protocol allows uncertainty about the global structure of the computation which can be resolved correctly at failure time, thereby reducing the overhead for failure-free execution.

We implemented two variants of optimistic finish, a centralized and a distributed implementation, and compared them to corresponding pessimistic resilient finish implementations in the X10 language. Performance results using micro-benchmarks, KMeans clustering application, and LULESH hydrodynamics proxy application show significant reductions in resilience overhead with optimistic finish compared to pessimistic finish. A resilient execution of LULESH over 343 places, recovering from a process failure using frequent checkpointing, acheived resilience overhead of 78% with optimistic finish, compared to 186% with pessimistic finish. Our proposed optimistic finish protocol is applicable to all task-based runtime systems offering automatic termination detection for arbitrary task graphs.

Authors' address: Sara S. Hamouda, sara.salem@anu.edu.au; Josh Milthorpe, josh.milthorpe@anu.edu.au,
Australian National University, Canberra, ACT, 2601, Australia.

1 INTRODUCTION

Recent advances in high-performance computing (HPC) systems have resulted in greatly increased parallelism, with both larger numbers of nodes, and larger numbers of computing cores within each node. With this increased system size and complexity comes an increase in the expected rate of failures. Programmers of HPC systems must therefore address the twin challenges of efficiently exploiting available parallelism, while ensuring resilience to component failures.

Dynamic task-based execution models present an attractive approach to both of these challenges. A dynamic computation evolves by generating asynchronous tasks that form an arbitrary directed acyclic graph. A key feature of such computations is termination detection (TD): determining when all tasks in a subgraph are complete. In an unreliable system, additional work is required to ensure that termination detection is correct even in the presence of component failures. Task-based models for use in HPC must therefore support resilience through efficient fault-tolerant termination detection protocols.

Fault-tolerant termination detection has previously been studied in Cilk-like spawn-sync models, in which each task must wait for the termination of its immediate children. APGAS languages like X10 implement a more expressive *async-finish* task model, which allows termination detection over sub-graphs of arbitrary depth. Previous research related to the async-finish programming model has mostly focused on productivity or performance, and ignored issues related to resilience. However, a resilient TD protocol was recently developed for the X10 language [2]. This protocol maintains a consistent view of the task graph across multiple processes, which can be used to reconstruct control flow in case of a process failure. In this work, we review the published protocol for X10, and demonstrate that the requirement for a consistent view results in a high performance overhead for failure-free execution. We propose an alternative protocol that relaxes the consistency requirement, resulting in faster failure-free execution with a moderate increase in recovery cost.

We begin with a review of dynamic task-based computation models and approaches to resilient termination detection. Next, Section 4 presents an overview of the Resilient X10 programming model; Section 5 reviews the pessimistic protocol used in previous implementations of Resilient Finish in X10; Section 6 presents our proposed optimistic protocol; and Section 7 evaluates the performance of the new protocol on a set of micro-benchmarks and two more realistic application benchmarks.

2 BACKGROUND: DYNAMIC COMPUTATION MODELS

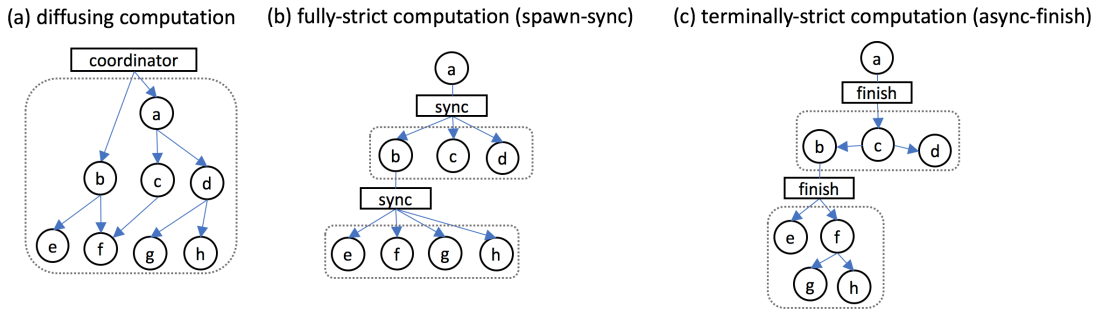

Fig. 1. Dynamic computation models that require termination detection. A dotted-box represents a single termination scope. Circles represent work units, which are processes in (a), and tasks in (b) and (c)

Dynamic computations can be classified into three main models. The standard model of termination detection is the diffusing computation model [3](Figure 1-a). In that model, the computation starts by activating a coordinator process that is responsible for signaling termination. Other processes are initially idle and can only be activated by receiving a message from an active process. An active process can become idle at any time. The computation terminates when all processes are idle, and no messages in transit to activate an idle process [13]. A diffusing computation has a single termination scope that is signaled by the coordinator process.

Computations that entail nested termination scopes are generally classified as fully-strict or terminally-strict [1] (Figure 1-b,c). Blumofe and Leiserson [1] describe a fully-strict task DAG as one that has spawn edges from a task to its children, and join edges from each child to its direct parent. In other words, a task can only wait for other tasks it directly spawned. On the other hand, a terminally-strict task DAG allows a join edge to connect a child to any of its ancestor tasks, including its direct parent, which means a task can wait for other tasks it directly or transitively created. Cilk's spawn-sync programming model, and X10's async-finish programming model are the most prominent representations of the fully-strict and terminally-strict computations, respectively. By relaxing the requirement to have each task to wait for its direct successors, the async-finish model avoids unnecessary synchronization while creating dynamic irregular task trees, that would otherwise be imposed by spawn-sync. Guo et al. [4] provide more details on X10's implementation of the async-finish model, contrasting it to Cilk's spawn-sync model.

From the diagrams in Figure 1, we can identify the following relations between the three models: 1) a diffusing computation is an async-finish computation with a single finish scope, 2) a sync-spawn computation is an async-finish computation where each finish is governing direct tasks only, and 3) a single spawn-sync scope is a 1-level diffusing computation. Thus async-finish is a super-set of both diffusing and spawn-sync computations, and in theory async-finish TD protocols are directly applicable to the other two models. Although a single finish scope can be considered as a diffusing computation, most diffusing TD protocols force parent/child synchronization constrains between tasks that make them identical to the spawn-sync model, and therefore not always applicable for async-finish computations.

When failures occur, nodes in the computation tree are lost, resulting in sub-trees of the failed nodes breaking off the computation structure. Fault-tolerant termination detection protocols aim to reattach those sub-trees to the remaining computation to facilitate termination detection.

3 RELATED WORK

Termination detection of a dynamic computation is a well-studied problem in distributed computing, having multiple protocols proposed since the 80s. Interested readers can refer to [7] for a comprehensive survey of many TD algorithms.

The DS algorithm, presented by Dijkstra and Scholten [3], was one of the earliest TD algorithms for diffusing computations and has been extended in different ways for adding resilience. In a computation that sends M basic messages between n processes, DS adds exactly M control messages to detect termination. The control messages are acknowledgements that a process must send for each message it receives. DS enforces the following constrains to ensure correct termination detection. First it requires each process to take as its parent the predecessor that caused it to switch from idle to active state (i.e. the origin of the first message it received while being idle). Second a child process must delay acknowledging the parent messages until after it has acknowledged all other predecessors, and received acknowledgements from its successors. By having each parent to serve as an implicit coordinator for its children, DS ensures that termination signals flow correctly from the leaves to the top of the tree. That also transforms the diffusing computation to a spawn-sync computation. Fault-tolerant extensions of the DS algorithm are presented in [5, 6].

Lai and Wu [5] describe a resilient protocol that can tolerate the failure of almost the entire system without adding any overhead for failure-free execution. The idea is that each process (idle and active) detecting a failure must detach from its parent, adopt the coordinator as its new parent, and share its local failure knowledge with its parent and the coordinator. On detecting a failure, the coordinator expects all processes to send termination signals directly to it. The protocol introduces a sequential bottleneck at the coordinator process when failures occur, which limits its applicability to large-scale HPC applications.

Venkatesan [13] presented a TD algorithm that relies on replicating the local termination state of each process on k leaders to tolerate k-process failures. The protocol assumes that the processes are connected via first-in-first-out channels and that a process can send k messages atomically to guarantee replication consistency. Unfortunately these features are not widely available in large-scale HPC systems, which limits the applicability of this algorithm.

Lifflander et al. [6] took a practical approach for resilient TD of a diffusing computation. Based on the assumption that multi-node failures are rare in practice, and that the probability of k-node failure decreases as k increases [8, 10], they designed three variants of the DS protocol that can tolerate most but not all failures. The INDEP protocol tolerates the failure of a single process, or multiple unrelated processes. It requires each parent to have a consistent view of its successors and their successors. To achieve this goal, each process notifies its parent of its potential successor before sending a message to it. Two more protocols were proposed to address related-process failures. In addition to the notifications required in INDEP, these protocols also require each process to notify its grandparent when its state changes from interior (non-leaf node) to exterior (leaf node) or vice versa. A failure that affects both an interior node and its parent is fatal in these protocols. Two special network services are required for correct implementation of the three protocols: a network send fence and a fail-flush service.

To the best of our knowledge, the only prior work addressing resilient termination detection for the async-finish model is that of Cunningham et al. [2]. They describe a protocol which separates the concerns of termination detection and survivability. Their protocol makes few assumptions about what parts of the system may be affected by a failure, however, they assume the availability of a resilient store that maintains TD-critical data. The survivability of the resilient store is the deciding factor in the survivability of the system. The proposed protocol uses six control signals to enable repairing the global structure: three finish-related signals, and three task-related signals. The added control messages ensure a consistent view of the entire finish scope, that facilitates recovery. However, they impose high overhead in the failure-free scenario. We refer to this protocol as 'pessimistic finish', since it tries to be well prepared for failures assuming they are common events. Unlike previous work, this protocol does not require any special network services other than failure detection.

Our work combines features from [6] and [2] to provide a practical low-overhead termination detection protocol for the async-finish model. Assuming multi-node failures are rare events, we propose a more message-efficient 'optimistic' protocol that significantly reduces the resilience overhead in failure-free scenarios.

4 X10: AN OVERVIEW

4.1 Programming Model

X10 is an asynchronous partitioned global address space language. It models a parallel computation as a global address space partitioned among places. Each place is a multi-threaded process with threads cooperating on executing tasks spawned locally or received from other places. Initially, all places are idle except place zero. It starts by spawning a task that executes the main method within a finish scope. This finish is the top of the task DAG, and is used for signaling final termination.

X10 programs dynamically generate arbitrary task DAGs by nesting **async**, **at**, and **finish** constructs. The **async** construct spawns a new task at the current place such that the spawning task (the predecessor) and the new task (the successor) can run in parallel. To spawn an asynchronous task at a remote place p, **at** is used with **async** as follows: (`at (p) async S;`). The **finish** block is used for synchronization; it defines a scope of coherent tasks, and waits for their termination. Exceptions thrown from any of the tasks are collected at the finish, and are thrown in a `MultipleExceptions` object after finish terminates. It is worth mentioning that **at** is a synchronous construct too, which means that `at (p) S;` (without async) does not return until S completes at place p. An error raised by S or while spawning S is thrown in an `Exception` object. Unlike **finish**, **at** does not wait for successor asyncs spawned by S.

A place may hold global references to objects hosted at remote places using the `x10.lang.GlobalRef` type. To access a remote object using its global ref `gr`, a place must shift to the object's home as follows: (`at(gr) gr().doSomething();`). This model enables moving computations to data in X10 programs.

4.2 Task Events

When a place creates a new finish block, it creates a (global) `FinishObject` that maintains the structure of the sub-tree managed by that finish. Each other engaged place creates a (local) `FinishObject` that maintains local TD-related state, and accesses its controlling global finish using its *global ref*.

The FinishObject provides, in addition to other events, three task events to track the changes in the global control structure as the computation evolves: f.fork(p, q), f.begin(p, q) and f.join(q), where f is the finish object (local or global) controlling the current task[1], p is the source place, and q is the destination place of a new task from p to q. Before spawning a new task, fork(p, q) is called at the source place p to notify finish of a potential new task to place q. On receiving a remote task, the destination place creates a local `FinishObject` if it does not exist. It then invokes the event begin(p, q) to request a permission from finish to execute the received task. If permission is granted, the task executes. When execution completes, the event join(q) is called at the destination place to notify task termination.

Figure 2 shows a sample X10 program, with the corresponding task DAG. The figure shows, in addition to asyncs and finishes, the implicitly created finish objects (global and local) at different places. The statements at lines 10 and 13 are added to highlight the happens-before relations that finish imposes between statements at different places. These relations must always be maintained, even in the presence of failures.

```
1 at (p1) async { /* a */
2     finish { /* f1 */
3         at (p2) async {  /* b */ }
4         at (p3) async {  /* c */
5             finish { /* f2 */
6                 at (p4) async { /* d */
7                     at (p5) async { /* e */ }
8                 }
9             }
10            printf('tasks d and e completed');
11        }
12    }
13    printf('tasks b, c, d, and e completed');
14 }
```

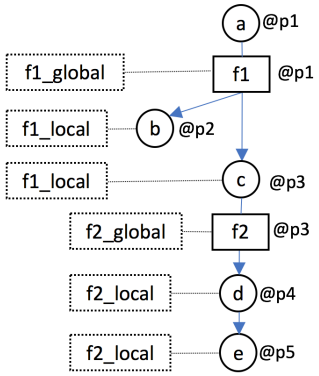

Fig. 2. A sample X10 program and corresponding task graph

4.3 Non-Resilient Finish Protocol

The default TD protocol in X10 is non-resilient and is fully asynchronous. To obey the asynchrony features of the async-finish model, parental responsibility is only given to finishes, not to tasks. In other words, all engaged places send their termination signals to their controlling finish, not to their parent tasks. A place sends a termination signal whenever the number of running local tasks reaches zero; it does not wait for spawned remote tasks, or even local tasks that were forked but have not started yet (i.e. tasks in between fork(p, p) and begin(p) at place p). The non-resilient finish protocol uses only one remote signal, namely 'terminate', which is sent by the join(q) event from a local finish to a global finish. The other events, fork(p, q) and begin(p, q), are local events that do not generate communication.

Protocol Variables: Each finish object has the following variables: `lc`, and `succ[0..n-1]`, where `lc` is a positive number representing the number of running tasks locally, `succ[q]` is an integer number of spawned successor tasks from the current place to `q`, and `n` is the total number of places. A global finish uses a third variable `latch` to block the task that created the finish until termination is detected. The initial value of `lc` is 1 in a global finish, reflecting an implicit task that executes the finish body. On the contrary, a local finish initializes `lc` to zero as it does not require an implicit task. `succ[0..n-1]` is initialized to zero.

Task Signals: On fork(p,q): `succ[q]` is incremented at p. On begin(p,q): `lc` is incremented at q. On join(q): both `lc` and `succ[q]` are decremented at q. Note that `succ[p]` at place p, which is the number of tasks that

[1] Appendix A (Figure 4) explains the invocation of the finish APIs using an example

p completed locally, will equal -m when p completes m tasks. When lc reaches zero, a local finish sends a terminate signal carrying succ[0..n-1] to the global finish. The global finish adds this array to its own array. The passed negative values (representing completed tasks at different places) reduces the number of waited-for tasks, until it reaches zero.

4.4 Resilient X10

Cunningham et al. [2] describe an extension to the X10 runtime system that allows the implementation of fault-tolerant systems. They address fail-stop process failures, in which a failure impacting a place stops it from sending or receiving any future messages. Tasks and data at a failed place are permanently lost, requiring runtime-level followed by application-level recovery actions to be taken to allow the application to successfully complete. Resilient X10 aims to localize the impact of a place failure to the neighbouring places interacting with it. When a place fails, other places should be able to communicate normally.

The core principle guiding the design of resilient X10 is the Happens Before Invariance (HBI) principle: *failure of a place must not alter the happens-before relation between statement instances at other places* [2]. Let us examine this principle in the context of Figure 2. Assume that place $p3$ died after tasks d and e were created. Now tasks d and e are orphaned because their parent finish $f2$ is lost. Suppose $f1$ ignored these tasks and terminated. That would allow line 13 to run in parallel with d and e, which breaks the happens-before relation that the program defined. There are two options to handle tasks d and e to keep the happens-before relation; either kill them or adopt them by their grandparent and let them complete (i.e. force $f1$ to wait for d and e). Because killing the tasks would leave the heap in an undetermined state, Cunningham et al. [2] designed their fault-tolerant termination detection protocol based on adoption.

Resilient X10 assumes that critical TD-related data are stored in a resilient store that survives failures. The design of the resilient store is orthogonal to the termination detection protocol, thus different stores (i.e. centralized/distributed, disk-based/memory-based, native/out-of-process) can be used. However, the survivability of the protocol implementation is limited to the survivability of the used store. In the rest of the paper, we use the term *resilient finish* to refer to a finish object stored in the resilient store that corresponds to the non-resilient global finish object. Local finish objects are non-resilient and can be lost safely. The proposed TD protocol assumes no special network requirements other than failure detection.

5 RESILIENT PESSIMISTIC FINISH

The resilient finish protocol presented by Cunningham et al. [2] aims to capture an accurate up-to-date view of tasks transiting between places, and tasks running at each engaged place. When a finish detects that a place has died, it adjusts its counts to ignore all running tasks at the dead place. Transiting tasks are more complex to handle. When the destination place dies, the transiting task is assumed to be dropped by the network, and can therefore be safely ignored by finish. However, when the source place dies, a transiting task may be in one of three different states: 1) not transmitted, 2) fully-transmitted and will be received by the destination, or 3) partially-transmitted and will be dropped at the destination.

The solution adopted by this protocol is as follows: finish will always drop transiting tasks if their source or destination is dead. A destination place must always seek permission from finish before starting a received task. Finish grants permission only if both source and destination are alive, otherwise, it denies the request. When a permission is granted, finish changes the task from the transiting state to the running (live) state. In a situation where a finish is lost, the parent finish extends its counter set to add the transit and live counters of its dead child. Despite losing the global finish object, its counter set is still available in the corresponding resilient finish object in the resilient store.

Protocol Variables: A global finish has three variables: id (a globally unique id for the finish, it is composed of the finish home place and a local sequence number), lc (local count) and a latch (for blocking the task until termination). The corresponding resilient finish object in the resilient store has the following variables: id, children, trans[0..n-1][0..n-1], live[0..n-1], where id is the same id of the global finish, children is a list of child finish ids, trans[p][q] is the number of registered transit tasks from p to q, and live[p] is

the number of running tasks at p. A local finish at an engaged place has two variables id (the id of its global finish), and lc (local count).

From the paper and the open source implementation, we determined that the protocol achieved its goals using six signals: three finish-related signals (publish, add_child, and release), and three task-related signals (transit, live, and terminate).

Finish Signals: Publish is sent from a global finish to the resilient store to create a corresponding resilient finish object. It initializes live[p]=1, where p is the finish home. This live task will be canceled by a terminate signal when the finish body completes. Add_Child is sent from a global finish to its parent's resilient finish to add itself to the parent's children list. Release is send from the resilient finish to the global finish to release the latch upon termination detection.

Task Signals: The transit, live, and terminate signals are sent to the resilient finish object by the events fork(p,q), begin(p,q) and join(q). On receiving the transit signal: trans[p][q] is incremented if both p and q are alive. On receiving the live signal: trans[p][q] is decremented, live[q] is incremented and permission is granted to the destination, only if both p and q are alive. Otherwise, permission is denied and the variables remain unchanged. When a task terminates, the join(q) event is called. It decrements the local count lc at place q and when that count reaches zero, it sends a terminate signal to resilient finish. On receiving the terminate signal: resilient finish decrements live[q]. When the summation of trans and live tasks reaches zero, resilient finish sends a release signal to the global finish to release its blocked task.

5.1 The Recovery Protocol

When the resilient store detects a place failure, it scans all resilient finish objects and updates them to reflect the new state of the system after the failure. It drops any transit task from or to a dead place, by resetting trans[dead][*], trans[*][dead]. It also drops all live tasks at dead places by resetting live[dead]. A parent finish adopts its dead children by adding their counter set to its own. If any of these changes result in terminating any finish, a release signal is issued. Note that the local and global finish objects at engaged places are not consulted during recovery; the information maintained in the resilient finish objects are sufficient to correctly update the global control structure.

Note that the resilient store needs to be an active store that can execute the above recovery logic, not just a passive data store. The same requirement is needed in our optimistic protocol as well.

6 OUR PROPOSED PROTOCOL: RESILIENT OPTIMISTIC FINISH

The main drawback of the pessimistic finish protocol is introducing high resilience overhead in failure-free scenarios. Having three communications with the resilient store for each task introduces significant overheads in communication intensive kernels, such as exchanging ghost cells in stencil computations. Although the number of created finishes are much smaller than tasks in most applications, having three signals for each finish is also expensive.

This paper's main contribution is designing a more practical message-efficient termination protocol for async-finish. Our protocol uses two finish signals (publish and release), two task signals (transit and terminate), and two recovery signals (count_received and count_children). In failure-free scenarios, the finish and task signals are sufficient for detecting termination; every published finish will always eventually be released, and every transitting task will always eventually be terminated. However, these four signals do not provide sufficient information to recover the control structure when failures occur. For example, by omitting the add_child signal, a parent finish does not know the identities of its children and whether any of them needs adoption. Also by omitting the live signal, a finish does not know 1) whether the transit task was transmitted, 2) whether it was received and is still running at the destination (could be blocked by a child finish there), or 3) whether it terminated at the destination but its termination signal is still in flight. We found that the information needed to resolve these uncertainties are already known either by the resilient store or by the destinations of the transit tasks, and thus can be queried at recovery time using two recovery signals to resolve the uncertainties.

Protocol Variables: A global finish has five variables `id`, `parent_id`, `origin`, `lc` and a `latch`, where `parent_id` is the id of the parent finish. The origin of the finish, is the source of the task that created the finish. For instance, if place `p` invoked (`at (q) async finish S;`), the origin of this finish is place `p`. The origin attribute is used for sending *simulated termination signals*, as we will explain in section 6.1. The corresponding resilient finish object in the resilient store has the following variables: `id`, `parent_id`, `origin`, `transOrLive[0..n-1][0..n-1]`, `sent[0..n-1][0..n-1]`. The value `transOrLive[p][q]` is the number of tasks that are either transiting from `p` to `q`, or live at `q`. `sent[p][q]` is an increasing number representing the number of tasks sent from place `p` to place `q`. A local finish at place `p` has five variables `id`, `lc`, `received[0..n-1]`, `reported[0..n-1]`, `deny[0..n-1]`, where `received[q]` is an increasing number representing the number of tasks received from `q`, `reported[q]` is an increasing number representing the number of received tasks from `q` that completed and whose termination signals were sent, and `deny[q]` is a boolean value that if true means that future tasks received from place `q` must be denied. The `sent` and `received` variables at the resilient finish, and at each local finish are used to resolve uncertainties about the transit tasks whose source has died, as we will explain in section 6.1.

Finish Signals: The finish signals (publish and release) are used in the same way as in the pessimistic protocol. A resilient finish is created with the publish signal, and it sets `transOrLive[p][p]=1` and `sent[p][p]=1`, where `p` is the finish home place. The registered `transOrLive` task will be cancelled by a corresponding termination signal at the end of the finish body.

Task Signals: The fork(p,q) signal issues a transit signal to the resilient finish object that carries the source and destination of the potential task. On receiving the signal: resilient finish increments both `transOrLive[p][q]` and `sent[p][q]` if both `p` and `q` are alive.

On receiving a remote task, the pessimistic protocol sends a live signal. However, in our protocol no signals are sent by the begin(p,q) event. A destination place starts any task it receives if the source of the task is not deniable (i.e. `deny[p]=false`). In a failure-free scenario, no place is deniable. The local finish object increments its its local count `lc` and also increments `received[p]` to memorize the total number of tasks it received from the source place. After that the task starts at its destination place, it eventually terminates resulting in invoking the join(p,q)[2] event by the local finish object. As usual, the join event decrements the local count `lc`, and when it reaches zero a termination signal is sent to the resilient finish object. To notify termination, local finish extracts the list of tasks that were received but not reported (i.e. `received-reported`), and sends termination signals for each of these tasks. It then sets `reported=received`. In practice, the termination signals are combined in one message to the resilient store. On receiving a terminate signal, the resilient finish decrements `transOrLive[p][q]`, and leaves the `sent` array unchanged for recovery purposes. It sends a release signal to the global finish when the summation of the `transOrLive` array reaches zero.

6.1 The Recovery Protocol

The features of our recovery protocol can be broken into three parts: resolving of transit tasks from a dead place, simulating termination of a lost finish task, and resolving of transit tasks to a dead place.

Resolving transit tasks from a dead place: If `transOrLive[dead][q]` is equal to `x`, how many of `x` are actually alive? To answer this question, the resilient finish compares the number of sent tasks from dead to q (say `s`) with the number of received tasks by q from dead (say `r`). Then it sets `transOrLive[dead][q]=s-r`, so that it only waits for termination signals from tasks that were actually received by q. To count the number of received tasks by q, the resilient finish issues a count_received signal to q that carries the finish id. This signal performs two actions atomically at the local finish object: it sets `deny[dead]=true`, and returns the number of received tasks. It makes the dead place deniable so that q does not receive future tasks from the dead place, in case any of them was delayed by the network. Our protocol generates a fatal error if the destination place q is dead.

[2]Implementation note: we modified X10's join event to carry two parameters join(src,dst), rather than one parameter join(dst). In the optimistic protocol, the termination signal must carry the source and destination of the terminated task, not the destination only. The reason is that the temination signal cancels a transit task, and each transit task is identifiable by both its source and destination.

Simulating termination of a lost finish task: In the pessimistic protocol, when a global finish is lost, its corresponding resilient finish is adopted by its parent, then it becomes inactive. When it receives further requests from its engaged places, it forwards them to its adopter. We apply a different approach in our protocol. We keep the resilient finish object whose home place died active (as a *ghost resilient finish*), and we allow it to handle task signals from its engaged places as normal. Hence, we avoid over-burdening the parent finish with signals from orphan tasks of its dead child. Moreover, we require a ghost resilient finish to simulate the termination of its lost enclosing task.

Normally, when a finish `f` terminates, its enclosing task eventually terminates, and sends a termination signal to `f`'s parent. The termination signal carries the source as the task's predecessor (origin) and the destination as the current place (which is also `f`'s home place). A ghost resilient finish has already lost its enclosing task, consequently the above termination signal won't be sent, and `f`'s parent will wait forever. To avoid this problem, we require a ghost resilient finish to simulate the termination of its enclosing task using its `origin` attribute. A simulated termination signal carries three parameters: a flag to indicate to the parent that this is a simulated termination signal, the `origin` of the ghost finish as the source, and the finish home as the destination. The finish home is obtained from its id.

Resolving transit tasks to a dead place: Transit tasks sent to a place that died can be in one of three states: 1) were blocked by children finishes at the dead place, 2) successfully terminated and their termination signals are in flight, or 3) lost with the place. To resolve this uncertainty, a resilient finish with id `x` sends a count_children request to the resilient store to count its children finishes (i.e. resilient finish objects whose `parent_id` equals to `x`), say the result is `y`. It then sets `transOrLive[p][dead]=y`. If `y` is zero, this indicates that all tasks at place `dead` were either completed and their signals are in flight (option 2), or were lost with the place (option 3). By setting `transOrLive[p][dead]=0`, we reflect the absence of any live tasks at `dead`. When the in flight termination signals reaches the resilient store, it drops them. If `y` is greater than zero, this indicates that some tasks were blocked by children finishes when the place died (option 3). To respect the happens-before-invarience principle, a child finish must terminate before a parent finish. By setting `transOrLive[p][dead]=y`, we force the parent finish to wait for `y` termination signals from its children, or more precisely, `y` simulated termination signals from its ghost children whose home place has died.

Our protocol fails to resolve the uncertainties if there is any transit task whose source and destination are both dead. The implementation is also limited to the survivability of the used resilient store.

7 PERFORMANCE EVALUATION

7.1 Protocols Implementations with Different Resilient Stores

The resilient store highly impacts the performance and the survivability of the implementation. In this paper we compare the performance of the pessimistic and optimistic protocol using two stores implemented natively within X10: a centralized store that saves all resilient finish objects at place-zero, and a distributed store that replicates each resilient finish object at two places. One replica is stored at the finish home along with the global finish object, and the second replica is created at the next place. The pessimistic finish protocol was evaluated using similar stores in [2]. However, the distributed implementation was later removed from the source code repository due to its instability. We identified a serious bug in the replication protocol used and hence re-implemented a corrected version of the distributed store for both the pessimistic and optimistic protocols. Interested readers may refer to Appendix-A for a comparison between the place-zero store and the distributed store. In the following results we use: (P-p0) to refer to pessimistic place-zero, (O-p0) to refer to optimistic place-zero, (P-dist) to refer to pessimistic distributed, and (O-dist) to optimistic distributed.

7.2 Environment Setup and Source Code

All experiments were conducted on Raijin supercomputer at NCI, the Australian National Computing Infrastructure, and NECTAR research cloud. Each compute node in Raijin has a dual 8-core Intel Xeon (Sandy Bridge 2.6 GHz) processors, and uses an Infiniband FDR network. We allocated 10 GiB of memory per node, statically bound each place to a separate core, and disabled hyperthreading. From NECTAR, we

allocated 29 virtual machines hosted on NCI nodes. These nodes have the same architecture of Raijin nodes, and are also connected by Infiniband. Each virtual machine has 12 VCPUs and 48 GB of memory. Our code is publicly available in the repository [12], forked from X10's main repository [14] revision number $3bc89e9$. X10 was built using GCC 4.4.7 for post compilation. We used MPI-ULFM (a fault-tolerant MPI implementation based on OpenMPI) as the transport layer of X10 on Raijin, and TCP sockets on NECTAR virtual machines. MPI-ULFM source [11] was built from revision $865e382$, which includes bug fixes to ULFM 1.1.

7.3 BenchMicro

Cunningham et al. [2] designed the BenchMicro program to measure the overheads introduced by resilient finish in different computation patterns, such as fan-out, all-to-all (or fan-out fan-out), and tree fan-out. The source code of the different patterns is included in the Appendix (Figure 5). We start all the computations from the middle place, not from place-zero as usually done in X10 programs. If we started the computations from place-zero, we would give an advantage to the centralized implementations since they will handle most of the signals locally. We recorded the average time for execution of each pattern from 5 invocations of the BenchMicro program, in which the pattern is executed repeatedly over at least 10 seconds, with at least 10 repetitions. We report times for each place count (256, 512, 1024) in Figure 3. The resilience overhead with 1024 places compared to non-resilient mode is shown in Appendix-A (Table 4). Our main conclusions from the results are as follows: 1) As expected, the optimistic protocols are successfully reducing the resilience overhead, and the percentage of reduction increases with the increase in the number of tasks. The only exception is the tree pattern (Figure 3-f), where O-p0 was unexpectedly slower than P-p0. However, it scales similarly to the other protocols. 2) Protocols that create big number of concurrent finishes, such as fan-out fan-out (fully-strict), and the tree fanout, scale poorly with a centralized store implementation, due to the sequential bottleneck of the store. For such computations, a distributed store is the most adequate option (see Figures 3-d,e,f).

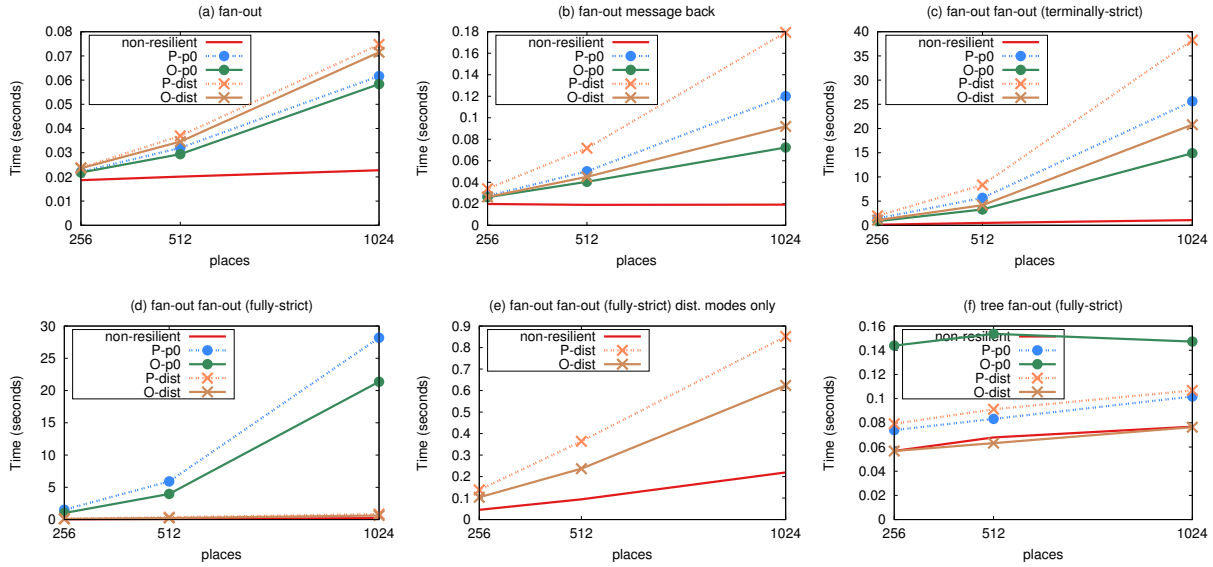

Fig. 3. Microbenchmark results. Figures d shows scaling for a fully-strict fan-out fan-out pattern, for all finish modes; figure e zooms in on the same results for distributed modes only

7.4 KMeans Clustering

We configured KMeans to find 300 clusters using 20 iterations for 100000 50-dimension points per place. We performed failure-free executions that does not involve checkpointing, as well as failure executions that checkpoint every 10 iterations and kills place $n/2$ after 30 seconds from program start. Killing a place resulted in re-executing 8 iterations with 64 places, and 6 or 7 iterations with 1024 places. We report the average total execution time of 5 runs in Table 1. The KMeans benchmark starts at place-zero. Each iteration performs a fan-out message-back computation, with each place assigning its points to different clusters, and reporting its local results to place-zero. The checkpointing framework used invokes a fan-out to each place to checkpoint its local data. Since KMeans is a compute-intensive kernel, the difference in the overheads between the different finish implementations is minimal. Based on figures 3-(a,b), we expected the fasted performance from O-p0, the slowest performance from P-dist, however, in the majority of cases, O-dist and P-p0 are the giving the fastest and slowest performance, respectively. We believe the source of the variation is due to stressing place-zero with computations as well as handling TD-signals. The results, however, still confirm the advantage of the optimistic protocol over the pessimistic one.

Table 1. KMeans performance on Raijin.

	64 places		1024 places	
	total time	overhead	total time	overhead
Non-resilient	71.38s	-	93.27s	-
P-p0	71.61s	0.32%	97.11s	4.13%
O-p0	71.42s	0.06%	96.85s	3.84%
P-dist	71.51s	0.18%	94.44s	1.26%
O-dist	71.40s	0.02%	93.82s	0.59%
With Failure P-p0	104.46s	46.35%	128.92s	38.23%
With Failure O-p0	104.31s	46.13%	128.84s	38.15%
With Failure P-dist	104.60s	46.53%	129.09s	38.41%
With Failure O-dist	104.34s	46.17%	126.46s	35.59%

7.5 LULESH

We now present results for LULESH shock hydrodynamics proxy application [9] with domain size 30^3, running for 30 iterations. In the failure scenario, we kill place $n/2$ just before iteration 15. The complexity of LULESH's code is much higher than KMeans. It involves point-to-point communication between neighbouring places for exchanging ghost cells, as well as collective operations. Both ghost exchange and collectives use `Rail.asyncCopy(..)` to perform one-sided get/put operations. The finish construct is internally used in different ways within these operations. Table 2 shows the performance of LULESH over 29 virtual machines from NECTAR research cloud[3]. LULESH requires a perfect cube number of places. Because each virtual machine has 12 virtual cores, the maximum number of places we could test was 343. The initialization kernel of LULESH is highly communication-intensive, each place interacts with all its 26 neighbors to obtain global access to remote buffers used for ghost cell exchange. With 343 places, the centralized modes (O-p0, and

[3]We experienced technical difficulties to run LULESH on Raijin over MPI-ULFM with the distributed implementations (O-dist, and P-dist). Errors raised because objects transfered between masters and backups were not serialized properly. We believe the complexity of LULESH along with the complexity of the replication framework stressed the runtime system in a way that activated a bug in the serialization framework. The bug was unavoidable on Raijin over MPI-ULFM. We could not fix it by the submission deadline, that is why we currently evaluate LULESH on NECTAR over TCP sockets.

P-p0) result in very high overhead for this kernel compared to the distributed modes. Overall, the optimistic distributed mode results in excellent performance in failure-free and failure scenarios.

Table 2. LULESH performance on NECTAR. Times are in milliseconds. Overheads shown in parentheses.

	64 places			343 places		
	init	step	total	init	step	total
Non-resilient	306	150	4824	1329	271	9481
P-p0	745 (143%)	198 (32%)	6688 (39%)	9123 (586%)	701 (159%)	30251 (219%)
O-p0	528 (73%)	186 (24%)	6103 (27%)	5328 (301%)	579 (114%)	22763 (140%)
P-dist	642 (110%)	231 (54%)	7588 (57%)	1906 (43%)	554 (104%)	18610 (96%)
O-dist	486 (59%)	168 (12%)	5527 (15%)	1653 (24%)	360 (33%)	12524 (32%)
With Failure P-p0	697 (127%)	196 (30%)	9076 (88%)	9245 (595%)	710 (162%)	44914 (374%)
With Failure O-p0	566 (85%)	186 (24%)	8150 (69%)	5283 (297%)	591 (118%)	32689 (245%)
With Failure P-dist	621 (103%)	231 (54%)	10239 (112%)	2171 (63%)	590 (118%)	27118 (186%)
With Failure O-dist	478 (56%)	162 (8%)	7178 (49%)	1585 (19%)	354 (31%)	16920 (78%)

8 CONCLUSION

We described *optimistic finish*, a termination detection protocol for the async-finish programming model. By reducing the signals required for tracking tasks and finish scopes, our protocol significantly reduces the resilience overhead of computation-intensive applications and micro-benchmarks, and enables them to reliably recover from the majority of failures.

REFERENCES

[1] Robert D Blumofe and Charles E Leiserson. 1999. Scheduling multithreaded computations by work stealing. *Journal of the ACM (JACM)* 46, 5 (1999), 720–748.

[2] David Cunningham, David Grove, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Hiroki Murata, Vijay Saraswat, Mikio Takeuchi, and Olivier Tardieu. 2014. Resilient X10: Efficient Failure-Aware Programming. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP âĂŹ14)*. 67–80.

[3] Edsger W Dijkstra and Carel S. Scholten. 1980. Termination detection for diffusing computations. *Inform. Process. Lett.* 11, 1 (1980), 1–4.

[4] Yi Guo, Rajkishore Barik, Raghavan Raman, and Vivek Sarkar. 2009. Work-first and help-first scheduling policies for async-finish task parallelism. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*. IEEE, 1–12.

[5] Ten-Hwang Lai and Li-Fen Wu. 1995. An (n-1)-resilient algorithm for distributed termination detection. *IEEE Transactions on Parallel and Distributed Systems* 6, 1 (1995), 63–78.

[6] Jonathan Lifflander, Phil Miller, and Laxmikant Kale. 2013. Adoption Protocols for Fanout-Optimal Fault-Tolerant Termination Detection. In *18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP âĂŹ13*. ACM.

[7] Jeff Matocha and Tracy Camp. 1998. A taxonomy of distributed termination detection algorithms. *Journal of Systems and Software* 43, 3 (1998), 207–221.

[8] Esteban Meneses, Xiang Ni, and Laxmikant V Kalé. 2012. A message-logging protocol for multicore systems. In *Dependable Systems and Networks Workshops (DSN-W), 2012 IEEE/IFIP 42nd International Conference on*. IEEE, 1–6.

[9] Josh Milthorpe, David Grove, Benjamin Herta, and Olivier Tardieu. 2015. *Exploring the APGAS Programming Model using the LULESH Proxy Application*. Technical Report RC25555. IBM Research.

[10] Adam Moody, Greg Bronevetsky, Kathryn Mohror, and Bronis R de Supinski. 2010. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In *Proceedings of the 2010 ACM/IEEE international conference for high performance computing, networking, storage and analysis*. IEEE Computer Society, 1–11.

[11] MPI Forum's Fault Tolerant Working Group. [n. d.]. MPI-ULFM Code Repository. https://bitbucket.org/icldistcomp/ulfm/. ([n. d.]).

[12] Sara S. Hamouda. [n. d.]. A Forked GitHub Repository of the X10 Language With Our Finish Implementations. https://github.com/shamouda/x10/commits/dist_finish_revamp/. ([n. d.]).
[13] Subbarayan Venkatesan. 1989. Reliable protocols for distributed termination detection. *IEEE Transactions on Reliability* 38, 1 (1989), 103–110.
[14] X10 Project. [n. d.]. The X10 language source code. https://github.com/x10-lang/x10. ([n. d.]).

APPENDIX A OPTIONAL MATERIALS

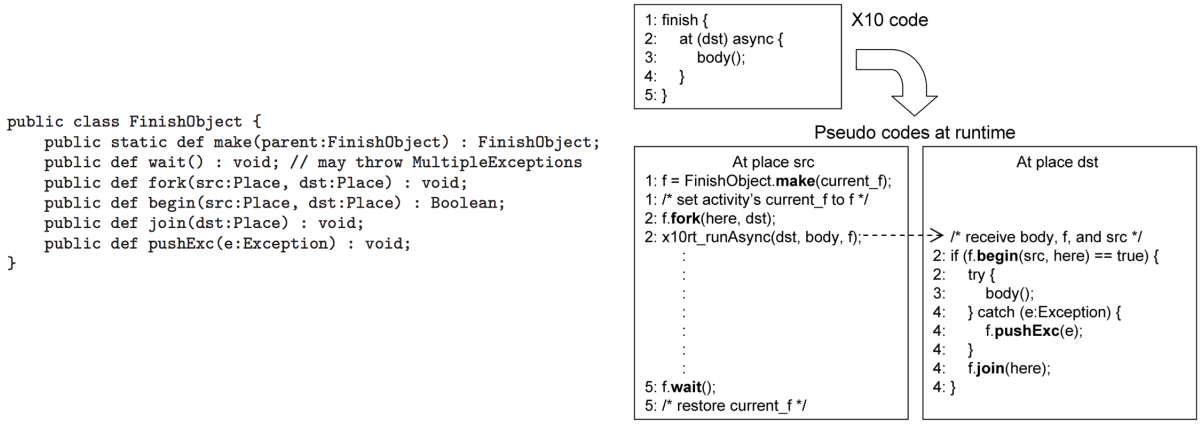

Fig. 4. Finish Runtime APIs. Source: Cunningham et al. [2]

Table 3. Comparison Between Place-Zero and Distributed Store (RF stands for resilient finish object)

	place-zero	distributed
Publish	creates a RF at p0	creates a master RF locally, and a backup copy at next place
Add_Child	done atomically with Publish	contacts the master and backup copies of RF's parent
Release	message from p0 to finish home	local release
Count_Children	done entiry within p0	a parent RF sends a message to the backup of the dead place
No Adoption Policy	noop	the backup RF recreates a new master copy of RF
Task Signals	one message to p0	two messages to master RF and backup RF
Fatal failures	loss of p0	simultaneous loss of master RF and backup RF

Table 4. Summary of BenchMicro Overheads at 1024 places

	fan-out	fan-out message-back	fan-out fan-out (terminally-strict)	fan-out fan-out (fully-strict)	tree
P-p0	170.92%	525.08%	2315.49%	12753.78%	32.51%
O-p0	156.44%	276.53%	1302.97%	9643.86%	91.60%
P-dist	228.39%	833.93%	3505.57%	288.45%	38.98%
O-dist	214.33%	378.65%	1858.71%	184.46%	-0.52%

```
1 val home = here;
2 // fan-out
3 finish { for (p in places) at (p) async S; }
4 //fan-out msg-back
5 finish { for (p in places) at (p) async { at (home) async S; } }
6 // fan-out fan-out (terminally-strict, one finish only)
7 finish { for (p in places) at (p) async { for (q in places) at (q) async S; }  }
8 // fan-out fan-out (fully-strict, a nested finish at each place p)
9 finish { for (p in places) at (p) async {
10     finish { for (q in places) at (q) async S; }
11 } }
12 //tree fan-out
13 def traverse() {
14    if (noChildren()) return;
15    finish {
16       at (getChild1(here)) async { traverse(); }
17       at (getChild2(here)) async { traverse(); }
18    }
19 }
```

Fig. 5. Code of BenchMicro Computation Patterns Used in Performance Evaluation