InContext: Simple Parallelism for Distributed Applications

Sunghwan Yoo[1], Hyojeong Lee[1], Charles Killian[1], and Milind Kulkarni[2]
[1]Department of Computer Science, [2]School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN
{sunghwanyoo, hyojlee, ckillian, milind}@purdue.edu

ABSTRACT

As networking services, such as DHTs, provide increasingly complex functionality, providing acceptable performance will require parallelizing their operations on individual nodes. Unfortunately, the event-driven style in which these applications have traditionally been written makes it difficult to reason about parallelism, and providing safe, efficient parallel implementations of distributed systems remains a challenge. In this paper, we introduce a declarative programming model based on *contexts*, which allows programmers to specify the sharing behavior of event handlers. Programs that adhere to the programming model can be safely parallelized according to an abstract execution model, with parallel behavior that is well-defined with respect to the expected sequential behavior. The declarative nature of the programming model allows conformance to be captured as a safety property that can be verified using a model checker.

We develop a prototype implementation of our abstract execution model and show that distributed applications written in our programming model can be automatically and efficiently parallelized. To recover additional parallelism, we present an optimization to the implementation based on *state snapshots* that permits more events to proceed in parallel. We evaluate our prototype implementation through several case studies and demonstrate significant speedup over optimized sequential implementations.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed and parallel languages*

General Terms

Languages

Keywords

Distributed programming, Mace

HPDC'11, June 8–11, 2011, San Jose, California, USA.

1. INTRODUCTION

Traditionally, there has been little need for parallelism in networking services such as distributed hash tables (DHTs); the majority of the execution time in a distributed system was spent in the application sitting on *top* of the networking layers, rather than in the networking layers themselves. However, network services are becoming more computationally complex, as they support features such as digital signatures, encryption and decryption, etc. Not only do such computationally expensive procedures consume processing time, they can often decrease responsiveness. As as a result, it is increasingly critical to take advantage of any parallelism or concurrency that might exist in network services, both to improve performance (by exploiting parallelism) and to improve responsiveness (by increasing concurrency).

Because network services process events coming from both the network and higher level applications, they have traditionally been written in an event-driven style. The Mace programming language [10] is a popular exemplar of this paradigm. In Mace, network applications are composed of a set of services. The lowest level services provide event handlers for networking events, and as part of their processing, they may invoke handlers of other, higher level services (these invocations are called *transitions*). The processing of a single event thus involves a series of transitions between loosely coupled services.

While the event-driven approach to writing network services is convenient, it does not lend itself to easy parallelization. Adding parallelism to an event driven program requires reasoning about all possible event handling paths (and hence, all possible chains of transitions) to determine which events might interfere with one another. To avoid such difficulties, Mace provides an *atomic event model*, where each event appears to execute atomically and in isolation. This is enforced with a single, global lock that is held while processing any event. While this approach is sufficient provided that events can be processed quickly, it becomes problematic once events become computationally expensive. First, a single global lock precludes any parallelism. Second, the lock can also be detrimental to system responsiveness; critical network maintenance messages (for routing or "heartbeat" purposes) may be delayed while long-running events are processed. Section 2 describes the Mace programming model in more detail.

Unfortunately, while abandoning the atomic event model might allow programmers to increase concurrency and exploit parallelism, it leads to a far more complicated and error-prone programming model. What is necessary is a

programming model that has the semantic simplicity of the atomic event model while still affording the performance of a model that abandons atomicity.

In this paper, we introduce the *InContext* programming model, an extension to the Mace programming language. InContext allows programmers to annotate transitions with *context* information, which specifies the expected behavior of the service. We provide a simple event model which governs how events can enter and leave contexts while processing a transition chain, and an execution model which allows events to be processed simultaneously. Crucially, the event model pertains to the behavior of a single event in isolation (and can hence be checked without reasoning about concurrent events). When programs written in compliance with the event model are executed according to the event model, InContext provides a straightforward guarantee: the network service, despite exposing concurrency and exploiting parallelism, will behave as if it executed in the atomic event model. The InContext event and execution models are described in Section 3.

In addition to the InContext programming model, we developed two tools to aid programmers in developing parallel network services. First, we formulated the InContext event model as a set of safety properties that can be checked by the Mace model checker [8], allowing programmers to verify that their applications adhere to the InContext model. Second, we developed an event-driven simulator that lets programmers estimate how much parallelism they might see from their network application. Section 4 discusses these tools in more detail.

To evaluate the InContext programming model, we performed four **(I hope!)** case studies to examine different usage scenarios, described in Section 7. First, we studied the use of InContext to improve responsiveness in a computational biology application. Second, we examined an application where InContext exposes parallelism and hence improves throughput. Third, we used InContext to provide an implementation of MapReduce [6] and evaluated its performance versus Hadoop [2]. Finally, we used InContext to implement a directory synchronization service, showing that InContext supports rapid development and deployment of parallel, distributed applications.

2. MACE

As described in the introduction, the Mace programming language allows programmers to write event-driven applications as a composition of independent services. These services are logically organized into an acyclic hierarchy, with the topmost services communicating with higher level applications (such as video players), and the lowest services communicating with the network. When an event is received, either from the network (in the form of a message) or from the application (in the form of a request), the appropriate service begins executing an event handler. This handler may invoke methods on higher level Mace services (called *up-calls*) or lower level Mace services (called *down-calls*) to complete its processing. These invocations are called *transitions*. A series of transitions that ultimately handle a message is called a transition chain, and though it was triggered by an event, as a shorthand we will call the transition chain itself an event.

For example, suppose a developer wishes to write a multicast service which provides verified message signatures on multicast messages. This functionality can be layered on top of the existing support for P2P tree multicast in Mace according to the structure shown in Figure 1. In this figure, we see that the Application U/I interfaces with the distributed system nodes by making calls (such as multicast) into a SignedMulticast service. The multicast transition is called a down-call because it is logically made from a higher-level component to a lower-level component. The SignedMulticast service is then responsible for signing the content, and calling the down-call multicast on the GenericTreeMulticast (GTM) service. GTM consults the Scribe [4] tree service to learn the structure by calling the down-call getChildren, then routes messages to multicast participants along the tree using the route down-call of the RecursiveOverlayRoute service, which in turn uses Bamboo [16] and the Mace-provided networking services to forward data to peers. This whole collection of nested transitions corresponds to a single atomic event in the Mace execution model, meaning developers need not worry about *e.g.* the tree being in flux while determining where to forward data to. Similar up-call transitions can be made from the transport services into the lower layers, eventually filtering up to delivering data back to the application, also within a single (separate) atomic event.

The Mace toolkit executes distributed systems implementations using a reactive event model. The application, TCP and UDP transport services, and supporting libraries such as the timer scheduler all internally utilize threads to perform I/O, blocking, and waiting, and then at appropriate times deliver events to Mace distributed systems by calling the transition functions of a service component. These transition functions (such as the multicast example above) are public member functions of the service APIs declared to be provided by each of the services.

2.1 The atomic event model

For correctness reasons, the baseline Mace implementation ensures that events are processed atomically by explicitly prohibiting parallelism using mutex synchronization. This is the *atomic event model*.

The atomic event model is simple to use because it provides conceptual simplicity, allowing services to synchronously deliver sub-events to other services. Alternative models such as SEDA [18] require that even within a single process, a component wishing to call methods on another component can only queue events to be processed by that component and receive responses through its own message queues. It cannot simply synchronously invoke methods on another component and receive a return value.

The atomic event model makes it much simpler to reason about system behavior, both for the developers and for development tools such as dynamic software model checkers, which commonly use exactly this I/O Automaton model for exploring the state space of systems. Similarly, allowing an event to synchronously execute nested transitions on other local service components allows event handlers to follow an imperative style of programming many developers find more natural. Additionally, avoiding these extra queued events avoids a number of context switches, which add overhead.

However, the atomic event model has its downsides. Network services often conduct bookkeeping operations, to maintain routing tables or keep track of which remote nodes are still accessible. These bookkeeping operations are constantly occurring in the background, and handling them in

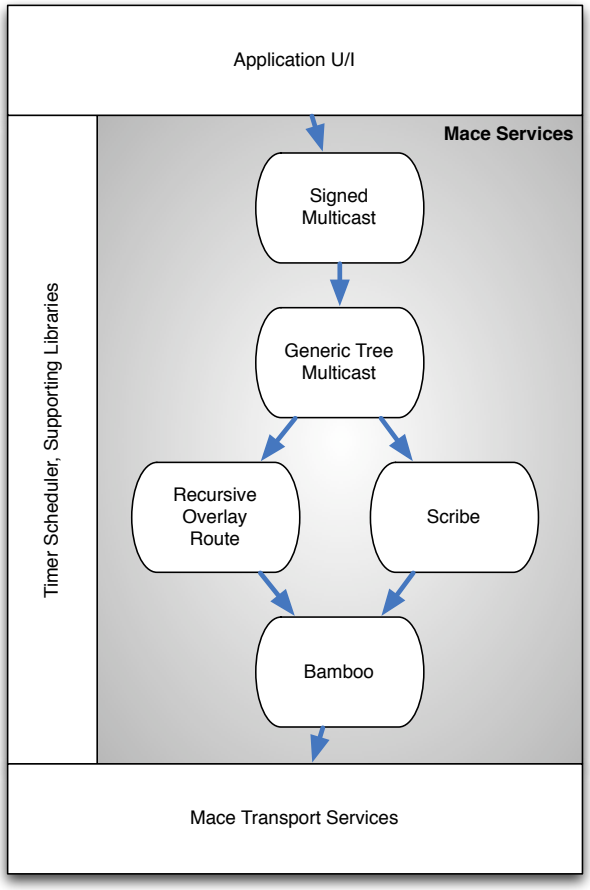

Figure 1: Architecture of a distributed multicast service in Mace.

a timely manner is crucial to the correct behavior of the system. Unfortunately, the atomic event model means that one long-running event will prevent these bookkeeping operations from being processed, reducing the responsiveness or even the correctness of the application. Furthermore, the atomic event model proscribes executing events in parallel, even if doing so could improve performance. As a consequence of the atomic event model, it was strictly forbidden to perform any blocking, waiting, or long-running computation as part of a Mace service.

A key contribution of this paper is overcoming the aforementioned limitation by enabling parallel event processing with only a small change to the programming model that preserves the event atomicity, and therefore the benefits of simplicity, model checking and using a high-level language for distributed systems development. While our implementation and experience is focused on the Mace development toolkit, the InContext event model itself is not tightly tied to Mace; the event model can be applied to other event-processing distributed systems frameworks, languages, and toolkits.

3. PROGRAMMING MODEL

Recall that an event in the Mace programming model is a series of transitions, triggered by the delivery of a message

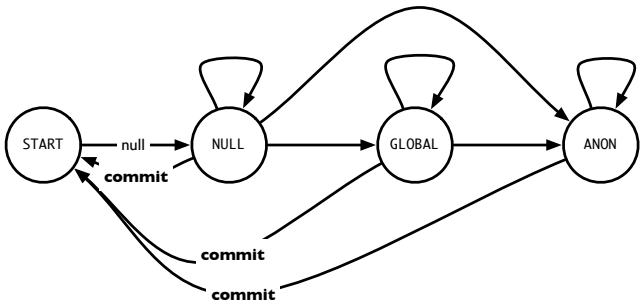

Figure 2: High level view of InContext programming model

(from the network to the Mace application), the expiration of a timer or a down-call from the higher level application. In the standard, sequential Mace model, these events are effectively processed one at a time, in the order in which they are initiated. The goal of the InContext programming model is to allow parallel execution of Mace events while preserving, as much as possible, the semantics of sequential Mace.

The InContext programming model is based on the notion of *contexts*, states in which an event can execute. The model consists of three types of contexts: global, anon and none. An event's actions in a particular context are restricted, as are the number of events that can simultaneously be in a context. For example, an event in global context can both read and write state, and only one event can be in global context at a time. The programming model also imposes a logical ordering on events. The following subsections describe the abstract event model (3.1) and the abstract execution model (3.2), which, taken together, allow us to place guarantees on the behavior of parallel execution (3.3).

3.1 Event model

An *event* in the InContext programming model is a series of transitions beginning with an initiating action (which can be considered a method invocation) and terminating with the return from that initiating invocation. An event can be in one of three contexts: global, anon and none. While an event is in a context, its behavior is restricted according to the rules of that context.

- An event in global context can both read and write the state of any Mace service. It can also initiate calls into the higher level application and await their return.
- An event in the anon context can read service state, but cannot write to it. While an event in this context can make calls into the higher level application, it can only do so if those calls are asynchronous (*i.e.*, they do not return a value).
- An event in the none context can neither read nor write service state. All of its operations must be based on "event-local" state. In other words, an event in none is functional in nature.

Events can send messages from any context (although, as discussed below, the message may be deferred). The contexts are hierarchically organized, according to their permissiveness:

$$\text{global} \succ \text{anon} \succ \text{null}$$

Events can transition between states according to the transition diagram given in Figure 2. When an event is initiated, it begins in the start state, and immediately transitions into the null context (the start state is for bookkeeping purposes). An event can *upgrade* from null to global context or to anon context. An event can also *downgrade* from global context to anon context. Note that once an event is in global or anon contexts, it cannot return to the null context except by taking a *commit* transition, the purpose of which will be explained in the next section.

3.2 Execution model

The event model above places restrictions on what an individual event can do in a particular context. The InContext execution model specifies the behavior of events as they interact with the system and with other events.

When an event is initiated, it is given a logical time stamp based on when it transitions from start state to the null context. In this manner, all events are placed into a total order. If an event takes a commit transition, returning it to the start state, it is logically divided into two *sub-events*, and the second sub-event is given a new time stamp. When a (sub-)event takes a commit transition, it pauses until all logically earlier (sub-)events have themselves committed. Hence, (sub-)events begin and end in order according to their logical time stamp.

The primary restriction placed by the execution model is in the management of the global context. Only one event may be in global context at a time. If an event wishes to upgrade from null to global, it must wait for any event currently in global context to downgrade to anon or to commit.

An event cannot enter global or anon context until all earlier events have either committed or themselves entered the anon context. This means that global context events act as fences: logically later events will not proceed until global context events move out of the global context.

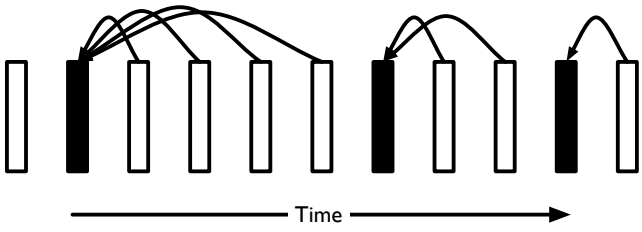

Figure 3: Sample event stream showing dependences between events

Recall that events in the anon context can read service state, but not write to it. The InContext execution model requires that events in anon see the state written by the last event with an earlier time stamp that entered the global context. Consider the stream of events shown in Figure 3. The events are ordered by time stamp, and black events are those that have been in global context. The dependence arrows show which state each event will read. Note that if an event downgrades from global to anon, the state it reads will be the state that it, itself, wrote.

Under most circumstances, events execute as they would in sequential Mace. They can take transitions and they can perform operations on service state (consistent with the context they are in). Certain actions, however, are treated differently in the InContext model. We call these *deferrable* and *non-deferrable* actions. Deferrable actions are those whose effects can logically be seen as happening at the end of an event's execution. These include sending messages on to the network and making calls to the application that do not return values. These actions are fundamentally asynchronous, and the behavior of an event does not depend on the results of these actions. The InContext execution model defers all such actions and executes them when the event (or sub-event) commits. Deferrable actions can occur in any context.

Other actions are non-deferrable; the only correct semantics are to perform the event immediately, before any further processing is performed. Examples of non-deferrable actions include calls into the higher-level application that return values used by the event and actions that query for user input. Non-deferrable actions, unsurprisingly, are performed immediately. These actions can only be performed while an event is in global context (recall that there can be only one such event). Immediately prior to performing the non-deferrable action, the event waits until all logically earlier events commit (hence executing when it is the only active event in the system).

3.3 Parallel execution guarantee

If all events adhere to the InContext event model, and a run-time system enforces the InContext execution model, we provide the following guarantee:

THEOREM 1. *The parallel execution of a Mace application whose events adhere to the InContext event model and whose runtime enforces the InContext execution model will produce the same results as if the sub-events were executed using the atomic event model, ordered by their timestamps.*

In other words, each sub-event will appear as if it executed atomically and in isolation, despite other events' potentially executing in parallel. Moreover, the overall execution will behave as if the sub-events executed sequentially in an order consistent with the sub-events' timestamps.

PROOF. This theorem is a straightforward consequence of the event model and execution model. The rules of the execution model can be seen as describing a read/write lock, where the lock must be held in write state while an event is in global context, and in read state while an event is in anon context. The event model makes clear that, for two events to interfere with each other, at least one must be in global context; the read/write semantics of the execution model thus prevent two events from interfering, providing the atomicity guarantee of the theorem. Downgrades complicate this somewhat: an event moving from global down to anon context can allow another event to enter anon context. However, because the first event cannot re-enter global context to modify state, atomicity is nevertheless preserved. The progress rules of the execution model prevent earlier events from being passed by later events, providing the ordering guarantee. □

Note that the behavior of sub-events means that this theorem does not imply that the parallel execution of a Mace

application will be precisely the same as the sequential execution. In a sequential execution, sub-events from a single event will occur consecutively, whereas in the parallel execution model, other (sub-)events can interleave between two sub-events that comprise a single event. Thus, while sub-events will appear to execute atomically, complete events consisting of multiple sub-events will not execute atomically. However, if events adhere to a stricter event model, where they only take a commit transition once, then the parallel execution will be exactly the same as the sequential execution. In other words, each event, despite executing in parallel, will appear to execute atomically.

4. ANALYSIS TOOLS

To support the InContext programming model, we have developed two analysis tools that allow programmers to check the correctness of their Mace programs and to analyze the amount of parallelism that exists in them.

4.1 Bug Detection Through Model Checking

Mace is packaged with a model checker [8] that can verify various safety properties of Mace programs. By exploring various possible transition chains, the model checker is able to guarantee that, for the explored portion of the state space, the safety properties hold.

A key advantage of the InContext event model is that an event can be checked for conformance with the model in isolation; no assumptions need be made about parallel execution or interleaving events. Furthermore, adherence to the event model can be encoded as a safety property that can be checked by the model checker.

We have extended the model checker to verify a number of properties. First, the model checker ensures that all (explored) events in a program only take transitions as prescribed by the event transition model of Figure 2. In particular, it ensures that events in anon context do not attempt to transition back to global context. While the event model allows events to take multiple commit transitions, the model checker can, optionally, verify that events only commit a single time. This allows programmers to verify that each event executes atomically. Second, the model checker can verify that, while in a particular context, an event adheres to the strictures of that context. In other words, it ensures that while an event is in null context, it only performs "functional" actions and while an event is in anon context, it only reads service state and performs deferrable actions.

By encoding adherence to the InContext event model as safety properties, the model checker provides programmers with a measure of confidence that their parallelized application will behave as expected.

4.2 Simulation of Parallel Execution

Mace is also packaged with a discrete-event simulator [9] which is used to evaluate performance and realistic behavior of a system implemented using Mace. Whereas the model checker is designed to exhaustively explore the system state space, teasing out potentially problematic configurations, the simulator is instead intented to explore the set of likely behaviors of the system by faithfully emulating timing behaviors of event durations and network latencies.

We have modified the Mace simulator to explore possible timings of the execution of programs developed using the InContext event model. The simulator was modified to keep track of, for each node, the last global event to commit and the last event (of any kind) to begin. By tracking these two events, the simulator allows the developer to explore the potential for parallelism in their application before deploying it in test-clusters or to end-users. Specifically, the modified simulator allows any new event to begin at its scheduled time (rather than waiting for the previous event to complete), provided that there are no pending global context events.

When used in concert with the model checker, the simulator allows for an intuitive development cycle. A programmer adds context annotations to his or her Mace program. The model checker is used to ensure that the annotated program adheres to the InContext event model. The simulator is then used to determine if the program will exhibit any parallelism when executed on the InContext execution model.

5. RUN-TIME SUPPORT

In this section, we describe the changes made to the Mace runtime system to support the InContext execution model. These changes consisted of three major components: (i) a *multithreaded transport* that allows a Mace application to receive and begin processing multiple messages from the network; (ii) a system of *context locking* that ensures that only one event is in the global context at a time, and that anon events read state from the most recent global event; and (iii) a commit pool that ensures that events commit in timestamp order.

These three components are supported through a set of ticketbooths and a read/write ticket lock. The typical processing of an event is as follows:

- The event is initiated due to a message received by the multithreaded transport. A thread from the thread-pool is assigned to the event, and it begins in start state (see Figure 2).
- The event transitions from start state to null context. In doing so, it enters a ticketbooth to acquire a ticket. The ticket represents the event's logical timestamp.
- If an event upgrades to global context, it acquires the r/w lock in write mode; if it upgrades to anon context, it acquires the r/w lock in read mode. If an event downgrades to anon context from global context, it downgrades the r/w lock from write mode to read mode.
- When an event prepares to commit (because it either explicitly commits or takes a commit transition), it checks its ticket against the ticketbooth. Until its ticket is "served," the event pauses. Once its ticket is served, the event performs all its deferred actions and releases the r/w lock, and then increments which ticket should be served (allowing the next event to commit).

The components that support this execution strategy are described in more detail below.

5.1 Multithreaded Transport

The native Mace transport utilizes a thread for socket I/O, which enqueues bytes read to be processed and eventually delivered by a single deliver thread in the transport. To enable parallelism of Mace programs, we modified the Mace Transport infrastructure to use a pool of threads or message

delivery processing. The deliver threads are responsible for delivering not just messages, but also socket errors and other network-related events to services. The deliver threads for each transport use a transport-specific mutex to synchronize dequeueing events, and then before releasing the transport mutex, they acquire a ticket for the event dequeued. It is critical that this ticket is acquired atomically with event dequeue, since in a distributed system, the event model must not violate the properties of the TCP transport - namely that messages are delivered in order. Many distributed protocols crash or fail when even a single pair of messages is delivered out of order. Note, however, that the ticket is only being synchronized with the other deliver threads of that specific transport. It is in a race to acquire the ticket against other transports, or other application components. This is acceptable since the there is no guarantee implied about the ordering of messages sent along different communication sockets, or of application behaviors with respect to arriving network messages.

5.2 Context locking

To implement the event model in Mace programs, transitions are annotated with the context in which they expect to be executed. In other words, transitions that expect to modify service state will be annotated with global, transitions that only read state will be annotated with anon, and transitions that are purely functional are annotated null. For backwards compatibility, un-annotated transitions are presumed to be global.

Every event tracks its current context (all events begin in the null context). When an event invokes a transition, it checks whether the context annotation is compatible with the event's current transition, according to the $\succ$ relation given in Section 3.1. If the context is compatible, no action is taken. Upgrades are implicit: if the transition requires upgrading to a higher-level context (*e.g.*, from null to anon), the event automatically upgrades its context. When an event upgrades, it acquires the r/w lock in the appropriate mode: write mode for global context, read mode for anon. To ensure that events enter contexts in the appropriate order, when acquiring the r/w lock in either mode the event must wait until its ticket is "served." Thus, events transition out from null context in the order they are initiated.

Downgrades in the InContext model are explicit: the programmer must insert a `downgrade` call. The programmer can specify whether the downgrade is to anon or null. Upon downgrade, the event checks whether the downgrade is legal (according to the transition diagram of Figure 2); if not, an exception is thrown. If the event downgrades to anon, the write lock is downgraded to a read lock. If the event downgrades to null, the lock is released entirely.

As soon as the event leaves global context (or, if the event never entered global context, as soon as it enters anon context), the "serving" number of the ticket lock is incremented, allowing the next event to acquire the lock. We note that the use of a ticket booth as part of the r/w lock also prevents writer-starvation, but observe that more important than avoiding starvation is the necessity to preserve the atomic event ordering seen in the equivalent execution.

5.3 Commit Ordering

When the transition is completing, the prior mode is again checked. If the current mode is not none and the prior mode is none, then the event is automatically committed, as the transition which entered the context is exiting. Additionally, an explicit downgrade to null counts as a commit (note that according to the transition diagram of Figure 2, this downgrade is a commit transition).

When threads are ready to commit, they first release the r/w lock, and then queue for the commit booth, where their tickets are examined again. Threads may only commit when their prior event has committed. Committing entails waiting until your turn to commit, then performing any deferred actions registered with the runtime. Deferred actions must be centrally queued, since despite the fact that they logically could have occurred at the end of the event, it is possible that their order matters (sent messages are deferred). Thus, the deferred actions must take place in the proper order.

If an event's prior event has already committed, this event is at the head of the commit queue (despite not attempting commit yet). In this case, we can say that this event has logically committed, and actions which can be deferred need not be deferred. Instead, the runtime will go ahead and deliver these actions immediately (first delivering any pending actions). Conversely, if the prior event has not committed, and a global event tries to perform a non-deferrable action, the runtime must block the current event until the prior event has finished committing. We note that this is possible since the r/w lock is released prior to committing (so the next event may proceed while the prior event waits to commit).

6. OPTIMIZATION: STATE SNAPSHOTS

The read/write lock implementation of the InContext execution model is attractive because it naturally enforces the restrictions of the execution model. Holding the write lock while in global context prohibits more than one event from being in global context, and holding the read lock while in anon context ensures that anon context events read state only from the most recent global context event.

Unfortunately, while the read/write implementation benefits from conceptual simplicity, it is susceptible to poor performance, especially in the presence of many global events that downgrade to anon context. When an event downgrades to anon, the InContext execution model allows another (later) event to enter anon context. However, because the earlier event still holds the read lock, no other events can enter global context. In fact, no non-anon events can begin until all currently executing anon events complete.

To address this bottleneck, we introduce an optimized implementation of the execution model based on *state snapshots*. As the name implies, this approach relies on snapshots of service state to ensure consistency while permitting additional concurrency. The snapshot-based model still employs a lock to protect global context, though it is now a simple exclusive lock, rather than a read/write lock. The difference between the snapshot runtime and the read/write runtime is in its handling of events entering the anon context. Rather than requiring that anon events hold the read lock, the first event to enter anon context after a global event (including the event itself, if it is downgrading) takes a *snapshot* of service state. This snapshot "freezes" the state as it was when the global event finished modifying state.

The snapshots are used by anon events to ensure that they read the proper state. When an anon event attempts to access service state, it instead accesses the snapshot as-

sociated with its most recent global ancestor. Because anon events read from the snapshots, later, global events can begin executing before earlier anon events commit.

To see how this can dramatically improve concurrency, consider an application that consists of events that spend a small amount of time in global context, downgrade to anon, and then perform a large amount of computation before committing. In the read/write implementation of the execution model, no concurrency will be exposed; every event needs to acquire the lock in write mode, and that cannot be done until all earlier events complete. However, in the snapshot model, each event will snapshot global state as it downgrades to anon, and continue operating on that snapshot. Having released the lock on the global context, the next event can begin. In this way, snapshotting enables far more concurrency.

There are a couple of issues that must be addressed when implementing snapshots. First is the expense of creating a snapshot. If there is significant state to record, the added time invested in taking the snapshot may overwhelm any concurrency gains. In the general case, this overhead can be mitigated through copy-on-write techniques (wherein the state is only copied into the snapshot when a later event modifies it). We do not implement copy-on-write, as we have found that service state tends to be small enough that snapshots are inexpensive.

The second consideration with snapshots is when to dispose of older snapshots. A snapshot must remain accessible until the last event that might access it completes. Because it can be hard to predict which events might access a snapshot, we adopt a conservative approach: when a global event commits, we know that all earlier anon events must have completed as well. Thus, we can remove the snapshot associated with the *previous* global event. Note that the possibility of state downgrades means that despite this garbage collection heuristic there may be multiple extant snapshots, each associated with a global event that has downgraded to anon, but not yet committed.

7. CASE STUDIES

We conducted four case studies of the InContext model. The case studies were intended to address different usage scenarios, and together demonstrate the generality and effectiveness of the programming model:

- Can InContext be used to improve concurrency? To evaluate this, we used InContext to implement a computational biology application. The original implementation, in baseline Mace, failed due to network timeouts and poor responsiveness. We compare that original implementation to a hand-written attempt to recover responsiveness to the InContext implementation. We show that the InContext implementation is easier to write and more effective than the hand-written implementation.

- Can InContext expose parallelism in applications where it exists? To evaluate this, we used InContext to parallelize a model application that simulates streaming data from multiple sources to multiple destinations. We demonstrate how the InContext model and its snapshot optimizations can be used to obtain a 7× speedup relative to the baseline implementation.

- Can InContext be used to build high-performance middleware for distributed applications? We use the InContext model to build an implementation of MapReduce [6], and compare its performance to Hadoop [2], a public implementation of MapReduce.

- Can InContext be used to rapidly develop and deploy distributed applications? We show how the InContext model can be used to develop a directory sharing service from scratch, and easily expose and exploit both parallelism and concurrency.

All of the evaluations were conducted on a cluster of 15 dual quad-core machines, each with 8GB of RAM and 4x1G Ethernet connections. In each case, 1 machine was set aside as the experiment controller, and did not take part in the experiment.

7.1 Computational Biology application

In prior work, we developed a calculation model to test the contingency of the coevoluted sites with using randomized binary contingency table approach [19]. We have been using the master-worker-model for sampling and analyzing binary contingency tables. It involves a series of rounds where a random portion of the input data is forwarded to each of the worker nodes, who in turn perform transformations on that data before returning it to the master node. This calculation can be done in parallel on a large dataset, providing some speedup of the BCT computation. The aggregation computation to combine results from workers takes, however, longer than the subdivided transformation process so we sought to develop an enhanced model.

Therefore we developed a new collaboration model which neither relied on a single master node, nor allowed the long-running summary computation to prevent the forward progress of the distributed computation.

However we have encountered a problem with this *baseline* implementation: the nodes sometimes fail to respond expediently when they receive ping messages to check liveness of one another. The particular reason of this temporary unavailability is caused by the calculation-specific transition which takes a long time until return. In the traditional Mace *atomic event model* it prevented nodes from handling delivered messages while running the calculation which sometimes led to reassigning a duplicate job to other nodes, or failure of the whole calculation when some nodes have concluded that the unresponsive node is completely failed.

Without the InContext approach to improving concurrency, we were consigned to the Mace atomic event model. To improve concurrency we used a *manual* re-implementation of the application. In this re-implemented application, the computation was divided into multiple sub-calculations, with periodic breaks to respond to network messages. With this modification in place, the node can stay responsive while finishing the computation. However, to implement this transformation we had to re-implement a substantial portion of the application, a tedious and error-prone process. Furthermore, the repeated context switching between handling network events and proceeding with the computation adds overhead.

In contrast, the InContext model allows us to easily modify the original implementation: computational events did not require modifying service state, and hence could be placed in the anon context. Similarly, most networking events are

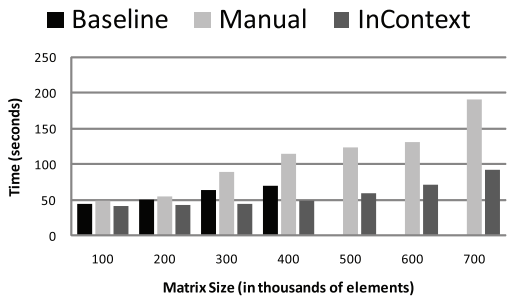

Figure 4: Running time of different variants of computational biology application vs. input size.

mere heartbeat events, and can also be executed in the anon context. By placing both types of events into anon, they can be executed concurrently, and the application can perform its computation while continuing to respond to networking events. Producing the *InContext* version of the code required few modifications to the baseline.

We compared the performance of the three versions of the computational biology application, using 7 different input sizes. The running time of each version for each input is given in Figure 4. As the input size increases, the length of the summary computation correspondingly increases. For input matrices larger than 500 thousand elements, the baseline version spends enough time in the summary computation that nodes cease responding to network messages and the computation terminates unexpectedly. While the manually implemented version avoids these timeouts, we note that its performance is significantly worse than either the baseline version or the InContext version, due to its frequent context switches. Finally, the InContext version demonstrates the strengths of our programming model: it maintains responsiveness even as the input size increases, and it outperforms both the baseline version and the manually modified version at all input sizes.

7.2 Streaming Application

To investigate the utility of the InContext programming model for exposing and exploiting parallelism, we developed a streaming application for signed, verified data streams based on the architecture shown in Figure 1. The application behaves just as described in Section 6 on the Snapshot optimization. First, the application nodes arrange themselves in a set of Peer-to-Peer multicast groups using the Scribe reverse-path-forwarding tree service and the Bamboo overlay routing DHT protocol. The application creates one group for each node in the system, and joins every group in the system. The application nodes then simultaneously wake up and multicast a message to their own group, to be received by all other nodes. These instances are repeated to conduct independent experiments to measure how long the delay is in delivering the data to each other node. The SignedMulticast service models message signature creation or verification, assumed to take 0.1 seconds. Because this signature creation would limit parallelism to the number of cores on a single machine, we simulate it by sleeping, allowing us to model larger systems given limited simulation resources. In our experiment, we run 4 instances of the application on each machine.

The data forwarding begins in the global context, since the lower-level forwarding services need to update tracking data and other state variables. Then, once the SignedMulticast service begins processing the data, the event is downgraded from the global mode. In the *ideal* case, the application would downgrade to the anon mode, so it may continue to read relevant state while processing the message, and ensuring that it is delivered in the correct order with respect to other events. However, as described in Section 6, this prevents parallelism from being realized when using the R/W lock implementation. To fully explore the potential performance of the InContext model, experiments were conducted using both snapshots and the R/W lock implementation, and also when the event committed (and began a new sub-event after the computation was complete) rather than simply downgrading to the anon context. This last choice would of course be appropriate for an application that could tolerate out-of-order message delivery, and also the potential that a message could be forwarded to a peer only became a peer after the message was initially received, that is, between message receipt and the sub-event where it is forwarded.

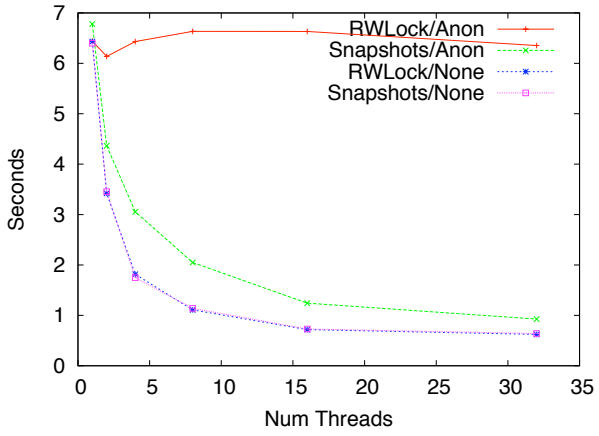

Figure 5: Synthetic benchmark application critical delays.

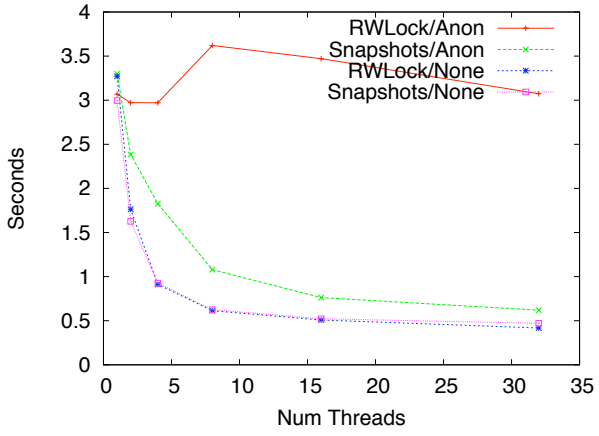

Figure 6: Synthetic benchmark application critical delays with respect to receipients.

The multicast experiment was repeated 10 times in each configuration, and was considered with a delivery thread pool containing from 1 to 32 threads. Thus we are able to see the potential speedup based on a wide range of the number of available processing cores. Figures 5 and 6 show two interpretations of the experimental outcomes. In the former, an experiment is considered complete when the last message arrives at the last recipient, that is, when all messages have been received. The plot shows, for each of the 4 configurations and the corresponding number of threads, the slowest experiment or the critical delay. In the latter, each receiver is considered separately, and receiver's result comes when it receives a majority of the messages. This case represents when an application is conducting majority voting, and can proceed when a majority of nodes have reported. We then report the slowest receiver for each experiment, and the slowest experiment for the graph.

In both cases, the trend is the same, though looking at the majority receipt case gives roughly half the values as the maximum case. When there is only 1 thread, the results are similar to if a R/W lock is used and downgrading takes place only to the anon context (allowing no parallelism. However increased threads allow the other three cases to achieve up to a 7X speedup, while the R/W-anon cases remain at their slow performance. Also, while committing the event and starting a new sub-event leads to some performance improvement over the other cases, it is modest over using snapshots and downgrading to the anon context, and decreasingly so as the number of threads increases. Note also that the R/W lock and snapshot cases are virtually identical if downgrading takes place to commit events. This is despite the snapshot case utilizing a snapshot when the sub-event is later created. Therefore, the cost of snapshotting for the synthetic application is unobservable, and using snapshots allows parallelism that the R/W lock alone could not.

We note that for InContext to deliver substantial performance improvements, the application must possess substantial parallelism to be exploited. In particular, events must spend significant amounts of time in either anon or null contexts; time spent in global context precludes parallelism. In an alternate version of the streaming app, without digital signing, there is little time spent in anon context, and InContext delivers no performance benefit. Notably, however, the InContext run-time system does not incur significant overhead, and the InContext implementation achieves performance comparable to the baseline Mace version.

7.3 MapReduce

We used the InContext model to develop a MapReduce implementation in Mace, called MaceMR. MaceMR is architected similarly to Hadoop, a popular MapReduce implementation that can take advantage of multicore systems: a single job tracker on the master node receives job requests, and sends tasks to nodes in the cluster. Each node has its own task tracker which assigns map and reduce tasks to various cpu cores.

MaceMR is a natural target for InContext. Map operations are purely functional, and hence can be treated as events in none context; InContext implicitly exploits any parallelism. We were able to develop MaceMR in one developer-month.

We implemented the benchmark application `wordcount` in MaceMR, as well as Hadoop. This benchmark splits a large

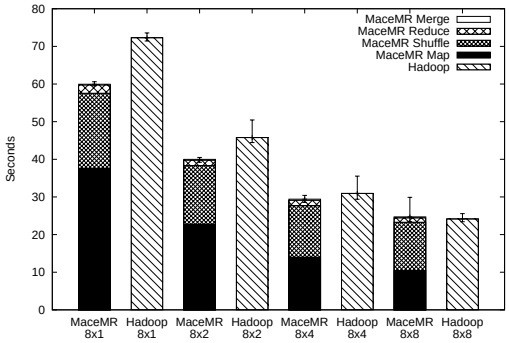

Figure 7: Total execution time vs. a set of NxT workers executing wordcount on a 100MB file. N nodes each with T threads. Results compare MaceMR to Hadoop, and illustrate that within MaceMC performance, scaling is achieved effectively during the map phase.

text file among the *map* nodes, each of which determine the frequency of each word in its portion. This data is then *shuffled* to the *reduce* nodes, which collate the information to determine word frequency data for the whole file. We ran `wordcount` in both MaceMR and Hadoop with 8 map nodes and 8 reduce nodes on a cluster with eight 8-core systems. We tried each variant with different numbers of threads (workers) on each node, ranging from 1 thread per node to 8 threads per node.

Figures 7 and 8 show the results of running the two versions of `wordcount` on 50MB and 100MB input files, respectively. For each configuration, we show the total runtime (averaged over 10 runs), and show how much time is spent in the various phases of the application. We see that, in general, MaceMR is well able to exploit the parallelism inherent in the map phase, scaling to 8 threads for both input files. However, because of un-optimized file I/O, neither the *shuffle* nor *reduce* phases exhibit much scaling, limiting overall scalability. Nevertheless, MaceMR has much lower overheads than Hadoop. Hence, for the small input file, MaceMR consistently outperforms Hadoop. For the larger input file, MaceMR is initially faster, while on larger numbers of threads, Hadoop's ability to parallelize the shuffle and reduce phases allows it to outperform our implementation.

7.4 Directory Synchronization

As our final case study, we investigated how much effort would be required to develop an InContext application from scratch. To do this, we built a rudimentary *directory synchronization service*, along the lines of DropBox [1]. The premise of this service is that a set of distributed nodes collaborate to keep a particular directory synchronized across all the nodes. Each file in the directory is coordinated by a single node, which is responsible for maintaining its canonical state. Each node is responsible for periodically checking the contents of its local version of the directory and propagating any changes in files to the appropriate coordinating node, which in turn distributes the change to all the nodes. As files are added and modified, the nodes must compute file hashes to determine if a file has changed; these hashes

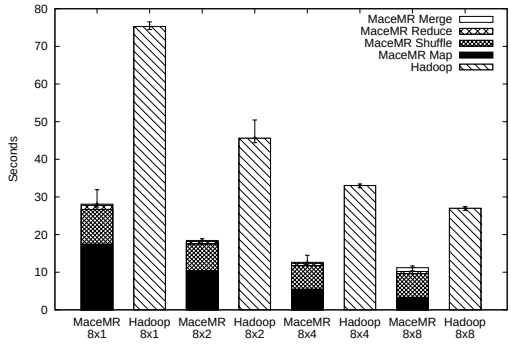

Figure 8: Total execution time vs. a set of NxT workers executing wordcount on a 50MB file. N nodes each with T threads. Results compare MaceMR to Hadoop, and illustrate that within MaceMC performance, scaling is achieved effectively during the map phase. Moreover, on the smaller input file, the lower MaceMR overheads cause it to perform consistently faster than Hadoop.

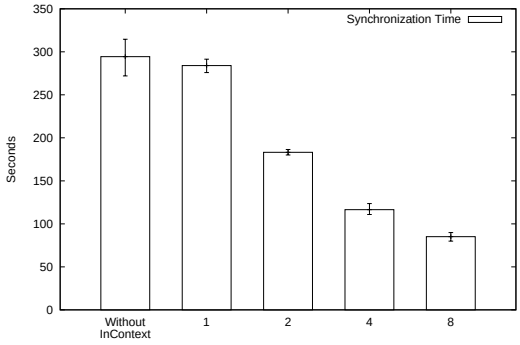

Figure 9: Directory synchronization time vs. number of event queue threads per node.

are propagated along with the file contents themselves, to ensure reliable and error-free delivery. To minimize network bandwidth, the application uses zlib to compress file contents while in transit, decompressing them before writing them to disk. Writing this basic application in Mace takes approximately 200 lines of code (in addition to the included Mace libraries that provide services such as hashing, routing, etc.).

Where does the parallelism arise in such an application? Files are independent of one another, and compressing, decompressing, and computing file hashes, particularly for large files, can be quite expensive. If multiple nodes are recording changes to the directory simultaneously, a remote node might receive many new files at once, and would need to compute hashes for all the files. This is an embarrassingly parallel procedure that nevertheless would be serialized in many models for writing distributed systems. This parallelism falls out naturally in the InContext: because the compression and file hashing transitions do not require writing service state, they can be performed in the anon context, and hence are naturally executed in parallel by the InContext runtime. The only transitions that must be executed in the global state, which precludes parallelism, are the initialization transition (for obvious reasons), and metadata update after receiving a file update, or scanning one locally. In the case of a file update, we divide the update event into two sub-events—the first, an anon event that prepares or computes state on disk, and the second, a global event that updates the node's metadata. Altogether, adding support for the InContext model to the baseline directory synchronization service required fewer than ten annotations.

We evaluated the application using both the baseline Mace implementation and the InContext implementation. The applications were executed on 5 dual quad-core machines, each running 1-8 event queue threads. To avoid disk bottlenecks, the directory was stored in a ramdisk. The directory was re-scanned every 60 seconds. We started the application at each node with a different set of roughly .9GB of data in 21 uniformly sized files, and measured how long the application took to install the last file on the slowest machine. Each experiment was repeated 5 times. Zlib compression, present in all experiments, reduced the network traffic by roughly 66%. The results can be seen in Figure 9. With only 1 thread, InContext behaves very similarly. As more event queue threads are added, performance scales nicely by leveraging parallelism for hashing, compression, and decompression. While some benefit may be seen beyond 8 threads due to systems overheads, the machines only have 8 cores, so further benefits should be minimal. At 8 event queue threads, the synchronization takes only 28% of its original amount.

8. RELATED WORK

Compiling sequential executions into parallel executions has long been researched in a variety of fields (*e.g.* [17, 7, 5]), and the best progress has been made in systems where developers specify a certain amount of implicit parallelism, allowing the compiler to parallelize the application in limited ways. One recent example of a tool for automated parallelism is the Bamboo [21] approach by Zhou and Demsky. This approach utilizes object parameter guards in a dataflow style to compile the language into locks which enable parallelism on shared memory multicore architectures. This is similar to our approach in that we are also providing implicit parallelism annotations to the compiler to compile into an appropriate locking scheme. However, the domain of our application, parallelizing event processing of a distributed application node, is substantially different, and different approaches are warranted. In the remainder of the related work section, we focus on the parallelism approaches used by toolkits used for building distributed systems.

SEDA. SEDA [18] is a staged, event-driven architecture which is used for building distributed systems. The BambooDHT [16] implementation was inspired by the design of SEDA. In SEDA, the implementation of a node is broken up into a set of components called *stages*, each of which is serviced by a pool of threads. Actions which spanned multiple stages were required to be broken up into a set of local events, since the mechanism for communicating across stages is to enqueue events in other stages. By enforcing a strict separation of stages, Bamboo is able to support parallelism with threads processing events independently at different

stages, requiring only synchronization with other threads in the same stage. The downsides of this model include the cost of context switches between threads for processing a request, and the more complex programming model.

P2. P2 [13] provides a dataflow language called Overlog for prototyping declarative overlays. Overlog is based on Datalog [3] and due to its dataflow nature, is capable of being naturally parallelized as rules execute upon input tuples, generate output tuples, and process these tuples without explicit control over the order in which the rules execute. Though the initially published framework did not perform this parallelization, it is an active area of exploration. Like SEDA, this approach is effectively parallelized, but requires programmers to adopt a new programming paradigm.

Splay. Splay [11] provides a very similar framework to Mace. Splay is focused on providing a unified framework for building distributed systems, rather than on supporting parallelism. It's event processing model is very similar to Mace, and provides no parallelism of event processing within a Splay process.

WiDS. Like Splay, WiDS [12] provides a similar framework to Mace. WiDS benefits from integration in Visual Studio, and is accompanied by a property checker [14] and replay tool. It's event processing model is very similar to Mace. What it does provide, however, is a limited amount of specialized parallelism provide by the framework. An example is a built-in heartbeat mechanism which can be provided by the framework rather than implemented as part of the service. In this manner, heartbeats will not be blocked by the processing of events. Note that for this approach to have solved our problem with the distributed computation experiment, it would have required application-specific heartbeats, verifying not just liveness, but also the current role of the node and progress in computation. Whether this parallelism is sufficient will depend entirely on the application developed in WiDS.

Libasync. Libasync [15] is a library for developing systems using asynchronous events. It is distinguished from toolkits like Mace and Splay in that it does not include a language component – its library calls are designed to be used natively in the C/C++ language. As such, it does not contain many of the additional components available for Mace such as the Mace Modelchecker. While originally a non-concurrent event framework, a follow-on version, libasync-smp [20], provided implicit parallelism. In libasync-smp, developers schedule an event with a specific event "color", and events of the same color will not be allowed to execute concurrently. Of the distributed systems frameworks, this model is closest to the InContext model we are providing. However, it does not support downgrading, wherein a thread executes in global context before moving to anon context. As we demonstrated in our results, this downgrading allows InContext to expose significantly more parallelism without affecting the nearly-sequential semantics of the model.

9. CONCLUSIONS

We presented an extension to the Mace programming language that allows programmers to write event-driven network services with a mental model of atomic events while ultimately allowing concurrency and parallelism in the application. This was accomplished through the InContext programming model, which couples an event model, which governs the behavior of individual events, with an execution model, which controls how events may be executed in parallel. The chief contribution of this paper is that applications which adhere to the InContext event model, and are executed using a run-time system that is compliant with the InContext execution model, will behave as if they were executed with the intuitive atomic event model.

We showed through a series of case studies that (a) the InContext model can expose parallelism in applications that possess any and (b) the InContext model can easily permit concurrency in applications that otherwise would lose responsiveness due to long-running events.

Finally, we developed tools to support the InContext model: a model checker that ensures that Mace programs adhere to the InContext event model, and a simulator that predicts the parallelism achieved by a Mace program running with the InContext execution model. By coupling these tools to an intuitive programming model with an efficient implementation, we feel that the InContext approach is a viable strategy for writing parallel distributed applications.

10. REFERENCES

[1] Dropbox. http://www.dropbox.com/.

[2] Hadoop. http://hadoop.apache.org/core/.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, and Antony Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal on Selected Areas in Communication (JSAC)*, 20(8), October 2002.

[5] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[6] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[7] Tim Harris and Satnam Singh. Feedback directed implicit parallelism. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 251–264, New York, NY, USA, 2007. ACM.

[8] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: Detecting liveness bugs in systems code. In *NSDI*, 2007.

[9] Charles Killian, Karthik Nagarak, Salman Pervez, Ryan Braud, James W. Anderson, and Ranjit Jhala. Finding latent performance bugs in systems

implementations. In *Foundations of Software Engineering (FSE 2010)*, 2010.

[10] Charles Edwin Killian, James W. Anderson, Ryan Braud, Ranjit Jhala, and Amin M. Vahdat. Mace: language support for building distributed systems. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 179–188, New York, NY, USA, 2007. ACM.

[11] Lorenzo Leonini, Étienne Rivière, and Pascal Felber. Splay: Distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *Networked Systems Design and Implementation (NSDI)*, 2009.

[12] Shiding Lin, Aimin Pan, Zheng Zhang, Rui Guo, and Zhenyu Guo. Wids: an integrated toolkit for distributed systems deveopment. In *Hot Topics in Operating Systems (HOTOS)*, 2005.

[13] Boon Thau Loo, Tyson Condie, Joseph M. Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. Implementing declarative overlays. In *Symposium on Operating Systems Principles*, pages 75–90, Brighton, United Kingdom, October 2005.

[14] Xuezheng Lui, Wei Lin, Aimin Pan, and Zheng Zhang. Wids checker: Combating bugs in distributed systems. In *Networked Systems Design and Implementation (NSDI)*, 2007.

[15] David Mazières. A toolkit for user-level file systems. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 261–274, Berkeley, CA, USA, 2001. USENIX Association.

[16] Sean C. Rhea, Dennis Geels, Timothy Roscoe, and John Kubiatowicz. Handling churn in a dht (awarded best paper!). In *USENIX Annual Technical Conference*, pages 127–140, Boston, Massachusetts, June 2004.

[17] Christoph von Praun, Luis Ceze, and Calin Caşcaval. Implicit parallelism with ordered transactions. In *PPoPP '07: Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 79–89, New York, NY, USA, 2007. ACM.

[18] Matt Welsh, David E. Culler, and Eric A. Brewer. Seda: An architecture for well-conditioned, scalable internet services. In *Symposium on Operating Systems Principles*, pages 230–243, Banff, Canada, October 2001.

[19] Sunghwan Yoo, Charles Killian, and Peter Waddell. Sampling and analyzing binary contingency tables: Implications for distributed computing models. Technical report, Purdue University, 2010.

[20] Nickolai Zeldovich, Alexander Yip, Frank Dabek, Robert Morris, David Mazières, and Frans Kaashoek. Multiprocessor support for event-driven programs. In *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX '03)*, San Antonio, Texas, June 2003.

[21] Jin Zhou and Brian Demsky. Bamboo: a data-centric, object-oriented approach to many-core software. In *PLDI '10: Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, New York, NY, USA, 2010. ACM.