Programming Model Support for Dependable, Elastic Cloud Applications

Wei-Chiu Chuang, Bo Sang, Charles Killian, Milind Kulkarni*
Department of Computer Science, *School of Electrical and Computer Engineering
Purdue University
{chuangw, bsang, ckillian}@cs.purdue.edu, milind@purdue.edu

Abstract

An attractive approach to leveraging the ability of cloud-computing platforms to provide resources on demand is to build *elastic* applications, which can scale up or down based on resource requirements. To ease the development of elastic applications, it is useful for programmers to write applications with simple, inelastic semantics and rely on runtime systems to provide elasticity. While this approach has been useful in restricted domains, such as MapReduce, we argue in this paper that adding elasticity to general distributed applications introduces new fault-tolerance challenges that must be addressed at the programming model and run-time level. We discuss a programming model for writing elastic, distributed applications, and describe an approach to fault-tolerance that integrates with the elastic run-time to provide transparent elasticity and fault-tolerance.

1 Introduction

One of the major promises of cloud computing is the ability to use resources-on-demand. Rather than build out a large-scale infrastructure to support a networked service with fluctuating needs, companies can build elastic components that start running on a very small infrastructure (perhaps only a handful of hosts), then scale up as the usage or popularity of the service grows. Elasticity also will allow changing infrastructure size on a smaller time-scale; monthly, weekly, or even daily variations in load could trigger the infrastructure size to change with the needs of the service. Further, elasticity could allow variable infrastructure size as a result of the availability of discount resources, or the sudden unexpected needs of the service in response to, e.g., flash crowds.

However, the development of elastic applications has been slower to occur, due to the complexity of developing high-quality elastic applications. One significant challenge of building such elastic applications is the difficulty of reasoning about application behavior, semantics, and correctness, when they are run across an unpredictable (and dynamically changing) set of resources. When running at larger scales, the likelihood of *some* part of the application failing increases, although the *fraction* of the application that fails is smaller. Applications need to be able to handle both the higher likelihood of failure, and ideally have techniques that can recover from such failures imposing a cost proportional to the fraction which has failed, rather than proportional to the size of the elastic application.

One solution to handling complexity of failure handling and failure semantics is for a programming toolkit to completely mask failures. Such approaches are readily available in distributed computing environments such as using MPI and data-flow computing, where failures can be ignored either by coordinated checkpointing and roll-back recovery [2], or by pairwise reliability protocols that perform checkpoint on send and careful message acknowledgement so as not to lose state [1]. The problems with such a "global" dependability approach are that (1) a one-size-fits-all approach to fault tolerance will inevitably not work for all situations. While these approaches are interesting and useful for special cases, they are often not broadly applicable. And (2), completely hiding faults from programmers has the side effect of eliminating an interesting class of applications that we anticipate might be worth writing in our model. Consider, for example, a scalable implementation of Paxos [14]. If the programming model hides all faults from the application, then there is little left to do for a Paxos implementation, and the behavior and fault handling is dictated by the programming model and its runtime system. Yet it should be possible to use a programming toolkit to produce an elastic version of Paxos that can run on as few as $2f+1$ physical nodes when there is little load, but can scale up to many more nodes to handle a large volume of simultaneous proposals from different applications. In doing so, however, we must not change the semantics or failure performance, lest the Paxos protocol be incorrect. That is, the scaled-up Paxos ought to execute on $2f+1$ *logical nodes*, where a logical node is composed of a group of physical nodes, while still being able to tolerate f simultaneous logical node failures. Work such as Fluxo [7] tries to seperate system functionality from architectural performance and scalability, which is similar to preserving semantics of an elastic application.

A few notable examples have arisen of successful elastic applications, successful largely because they can leverage a collection of well-understood patterns or application properties that simplify the challenges of partitioning, fault tolerance, and recovery. For example, computational applications, such as those based on MapReduce [3] or more generic batch scheduling [12], typically have little-to-no communication between computational partitions, and moreover support the luxury of simply recomputing any result which is lost due to a failure. The partitioned computation can temporarily store partial results (inputs to any stage), and only the failing stages need to be recomputed. Further, since only the final result is externally observable, the failure can easily be masked as internal stages are known to be "private" to the computational system.

In this paper, we propose a new programming model that supports elasticity and preserves both the semantics and failure characteristics of the developer's inelastic design. By doing so, we allow the developer to reason about the basic application design, while ignoring the complexity of elasticity. Our programming model promises to preserve the illusion that an application is running on a fixed set of resources (logical nodes) whose *throughput* can vary as the actual set of resources (physical nodes) changes. Our paper is based on the insight that many applications naturally decompose into a hierarchy of different *contexts*, namely that modular and component design and object-oriented programming have led to programmer designs that large sections of code are inherently bound to a subset of application state, and that the application state is often further bound to the parameters of the function calls. For example, as we illustrate in § 3.1, an elastic design of Conway's Game of Life [4] can be composed of contexts for each cell, and when computing the cell value for each iteration, you need only look at the state of neighboring cells, and not the global state of the system. Since our programming model is a simple annotation language on top of an existing distributed systems toolkit, and since the applications we target already naturally decompose into contexts due to good programming design, most applications can support elasticity through a simple transformation and code annotation that in our experience often further simplifies the application development thanks to the promotion of contexts to a first-class programming citizen.

2 Fault-tolerance in event-driven systems

There are three broad approaches to providing fault tolerance in distributed, event-driven systems. Algorithmic, node-level and process-level. In algorithmic fault tolerance, the program itself contains logic for handling failures. For example, the program might use Paxos [14] or other consensus protocols to tolerate the failure of nodes in the application. The key characteristic is that the logic for handling failure exists at the application level, and hence operates at the level of a logical node.

Rather than providing fault tolerance in the application logic, an alternate approach is to build fault-tolerance capabilities into the run-time system. An example of such an approach is MaceKen [16], which uses message logging to tolerate the failure of (logical) nodes in arbitrary event-driven programs written in the Mace programming language. Crucially, MaceKen relies on semantic properties of Mace programs that only exist at the logical-node level. Many MPI checkpointing libraries [1, 2] can be seen in a similar vein, relying on the semantics of the MPI API, which also operates at the logical-node level.

The problem with both algorithmic approaches and node-level approaches is that they operate at the level of logical nodes. However, in an elastic application logical nodes may be comprised of multiple physical nodes. The failure of a physical node, then, represents only a *partial* failure of the logical node (in a hardware context, this is analogous to a single core failing in a multicore system). Unfortunately, elasticity is transparent to the programming model, and hence there is no visibility of physical nodes at the application level. The best that could be done, then, is to treat the failure of a single physical node as the failure of the *entire* logical node that the physical node is a part of. This both dramatically slows down the restart process (as the entire logical node, which may have been distributed across many physical nodes, must be restarted) and increases the exposure to faults (as the failure probability of a logical node increases with the amount of elasticity).

One alternative is to provide fault tolerance at an *even lower* level. System-level checkpointing systems (*e.g.*, [6, 13]) checkpoint a process's state, and can migrate it to a new system to restart. Because this operates at the process level, system-level checkpointers could be used to provide fault tolerance for physical nodes. A major drawback to these systems, though, is that their checkpoint overhead can be large: they capture a process's entire state, rather than only that which is required to restart execution.

The inescapable conclusion is that providing effective fault tolerance for elastic applications requires a fault-tolerance system that integrates tightly with the system used to provide elasticity[1]. By doing so, the fault tolerance system will have access to individual physical nodes, permitting per-physical-node fault recovery, but will also have information about the program itself, allowing checkpointing to leverage application characteristics to avoid whole-process state saving.

[1]Note that this is essentially how models like MapReduce provide fault-tolerance: the fault-tolerance capabilities are built into the elastic runtime and enabled by the programming model

```
state_variables{
  NodeAddress printer;
  context Cells<int x, int y> {
    int iteration;
    bool val;
    PeerValues[][] pv;
  }
}
transitions {
  async storeVal(bool v, int x, int y, int px, int py)
      :: Cells<x,y> {
    pv[px][py] = v;
    if( /* pv is full */ ){ async_doCompute(x,y); }
  }
  async doCompute(int x, int y)
      :: Cells<x,y> reads Global {
    iteration++;
    val = f(pv, val); //updates cell value
    async_storeVal(val,x+1,y,x,y);
    async_storeVal(val,x+1,y+1,x,y);
    ...
    send_message(printer, Val(iteration, val));
  }
  ...
}
```

Figure 1: Pseudocode illustrating part of a design of Conway's Game of Life in our programming model.

What is especially interesting is that adding fault tolerance to an elastic run-time system need not be onerous. The following section explains our approach to elasticity, and § 4 argues that fault tolerance can be readily supported by repurposing many of the mechanisms used to support elasticity.

3 Elastic Programs

This section describes our context-based approach to writing elastic programs. The key goal of the approach is that the programming model should have simple, sequential semantics and should leave the difficulties of adding elasticity to the compiler and runtime system.

3.1 Programming Model

We are building our programming model on top of an existing distributed systems toolkit: Mace [10]. Mace already supports building a wide variety of distributed systems using a simple, messaging-based atomic event model or actor-like model. Mace has previously been shown to lend itself to the development of model checkers [5, 8], performance checkers [9], and tools to discover potential malicious insider attacks [11, 15]. Importantly, our programming model will preserve all necessary semantics of existing Mace programs so that in addition to the benefits of dependable elastic programming, these existing tools can be used with confidence to further test the robustness of systems built in our programming model.

Our new programming model is built on top of the InContext model [17], which introduced the basic notions of context that enables parallel execution of atomic events in Mace. In the InContext model, state is either

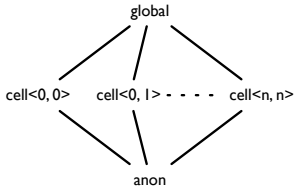

Figure 2: Sample context hierarchy for Game of Life

global and shared by an entire logical node, or *anonymous*, and local to a particular event. Annotations on methods in the program specify which state can safely be accessed, and the runtime system can use these annotations to determine whether events can run in parallel without breaking atomic event semantics, or must run in isolation. While InContext uses a coarse partitioning of state, it suffices in limited settings to provide parallelism.

Our model extends this to more fine-grained state contexts, and elastic, distributed event execution. A full language description of the programming model is beyond the scope of this paper—in this section we present some sample pseudocode (Figure 1), which we will use to explain the contexted execution model.

Figure 1 shows sample code that could be used in implementing Conway's Game of Life [4], and figure 2 shows the context hierarchy. The Game of Life is a simulation of a simple cellular automaton, where in each iteration, each cell in a matrix is either alive or dead. A cell's state in the next iteration is a function (f in the code) of its current state and the state of the surrounding cells.

In this implementation, there are two logical nodes: a compute node (shown in the figure), and a "printer" node that displays the results to the user. The compute node's state is hierarchically arranged into a *Global* context and a set of *Cells* contexts parameterized by coordinates x and y, each of which holds the state of a single cell.

Execution in our model is organized around *events*. Events are triggered either by the receipt of messages or by special *async* calls that begin new events on the same logical node. An event is begun by executing a *transition* (in our example, there are two transitions), which can then synchronously call other transitions as part of the same event. The event completes when the initial transition returns. The entire execution is logically atomic. This *atomic event execution model* makes it easy to reason about program behavior, and enables the additional testing tools mentioned previously.

Each transition executes in a particular context, which governs which state it has write access to. When an event invokes a transition in a context c, it "acquires" c by requesting an exclusive write lock at c. It can then invoke transitions in contexts c' that are *lower* in the state hier-

archy than c (so if an event is executing a transition in the global context, it can invoke a transition in a *Cells* context, but not vice versa). Events can only write to contexts they have acquired, though they can *read* from higher level contexts.

In this example, one iteration of the Game of Life would proceed by invoking DOCOMPUTE on each cell in the game. Note that each call to DOCOMPUTE is bound to a particular *Cells* context by its arguments. Thus, calling DOCOMPUTE(1,2) executes the method *in the context* of $Cells < 1,2 >$. DOCOMPUTE computes the cell's new value, then triggers new events in its neighboring contexts with asynchronous calls to STOREVAL, which update the values of the neighboring cells. When a cell receives messages from all of its neighbors, it triggers a DOCOMPUTE event again, beginning the next iteration. DOCOMPUTE also sends a message to the printer node to update its display; this message will trigger an event handler at the other logical node.

Importantly, this execution model is independent of any elasticity or distribution that may take place. It is defined in terms of a sequence of atomic events executing on a single logical node, and can easily be executed as such. As we will see in the following sections, this syntax and execution model allows both elasticity and dependability, while still providing the atomic event semantics programmers are used to.

3.2 Elasticity

Using the above programming model, a developer writes a program to execute on N logical nodes. Recall from the discussion in the Introduction that we specifically do *not* want to eliminate all distributed programming from the developer, as there will be cases and reasons for specifically programming the interaction of distributed components. In the sample code, for example, the code uses two logical nodes, because the *printer* specifically needs resources that are not available on the elastic computing infrastructure (namely, the user's display).

However, given an implementation written for N logical nodes, we would like to execute the events on M physical[2] nodes for some $M > N$, with the developer worrying only about the interaction protocol between the N logical nodes, and thus with the semantics of N logical nodes. The programming model just described naturally supports this elasticity. We map different contexts onto different physical nodes. An event executes on the physical machine that stores that transition's context. If it calls other transitions whose contexts are on other physical machines, these will be mapped into RPCs.

Contexts are the smallest unit of partitioning, and may be grouped on physical nodes as desired for performance reasons, such as to limit the costs of RPCs. Ideally, the mapping of contexts to nodes will have the property that no physical node's cpu, memory, network, or other resources are overloaded due to the load offered to the contexts on that node. To accomplish that goal will involve approaches such as co-locating closely related contexts, or locating them on topologically close machines when they must be split due to load. Conveniently, the hierarchical state model also provides strong indications as to the relationships between the contexts.

To see how to execute this programming model elastically while preserving atomic event semantics, we can consider each event as a transaction. We assign each event an event number when it begins, and require that the events commit in order. In one possible implementation, we designate a "head node" to be responsible for ordering all events in a single logical node. Co-locating the head node with the *Global* context is sensible, as that context is the common ancestor of all contexts, and thus could serve as a bottleneck for parallel processing. It therefore behooves a developer using our programming model to spend as little time as possible writing to the *Global* context. Rather than execute events using optimistic concurrency, we leverage the known commit order and the programming model to prevent transactional conflicts. Event transitions may only execute in one context, and may only make calls to enclosed contexts, so it suffices to block future events from executing as long as a prior event still is executing in a context that is or encloses the context of the future event. If a transition requires an ancestor's state, it will obtain a snapshot of the state that conforms to the transactional semantics—it includes the state of all transactions that will commit earlier, and none of the state of transactions that will commit later. Therefore, rollback can only occur as a result of failure, as described in the next section.

For any messages sent, they can be forwarded to the head node, which will transmit them in the correct order when the event eventually commits. Since the head node orders events, all messages are directed to each logical node's head node. The head node thus must act as an application-layer switch, and must be engineered for high performance or risk becoming a bottleneck. Since all incoming and outgoing communications are filtered through the head node, it will look to the external observe as the one node of the logical node.

A common challenge in elastic applications is the tension between finer-grained work division and the overhead of scheduling that work. Our design does not eliminate this tension, and as with all solutions may require some effort to determine the right granularity of division. Our prototype head node implementation can forward events at a rate of 10,000 per second, which should be suitable for many practical applications.

[2]In a Cloud setting, this is likely a virtual node. We use the term physical to distinguish it from the logical nodes a developer programs.

4 Dependability

As a logical node's contexts are distributed across more physical nodes, its exposure to failures increases. If a failed physical node triggers a logical node failure, the failure rate of logical nodes (and hence, the application itself) will increase. We thus would like to prevent physical node failures from causing logical node failures whenever possible[3].

Interestingly, we find that most of the machinery necessary to provide physical-node fault tolerance already exists in the elastic runtime system of the context model. Our elastic runtime system needs to support distributed, parallel execution of events. To do so, we implement three features: (i) events execute transactionally and in sequential order to preserve atomic-event semantics; (ii) events take snapshots of contexts as they execute (to allow multiple concurrently-executing events to access the same context in isolation); and (iii) all externally-visible communication is handled by the "head" node, with incoming messages triggering events, and outgoing messages deferred until event commit. These features directly enable adding physical-node fault tolerance to our system, with only minor modifications.

Consider what happens when a physical node with some context c fails. Because events access contexts in order, when the node fails, all events can be categorized into three groups: (i) events that have committed; (ii) events that have accessed c and taken a snapshot of c and will not need write-access again; and (iii) events that currently have write-access to c or have not yet reached c. The sequential ordering of events means that all events in group (i) are logically before all events in group (ii), which are logically before all events in group (iii).

The transactional execution of events means that only the events in group (i) have affected the externally-visible state of the logical node. Thus, to recover from this failure, we can (a) terminate all events in groups (ii) and (iii); (b) re-map c to a new physical node and restore it with the state as it existed for the latest event in group (i); (c) restart all the events in (ii) and (iii).

There are two tricky pieces to the above protocol. First, we must be able to restore c's state. Recall that events naturally take snapshots of context state as they execute. As events commit, any snapshots of contexts they took can be replicated to other physical nodes. Note that we need only save the most recent committed snapshot for any context. This set of committed snapshots is a checkpoint of the logical node's committed (and hence externally visible) state. Upon failure, the head node can refer to the replicated snapshot to restore a physical node's state. Note that the cost of restoring this state is proportional to the size of the contexts on the failed node, not the size of the overall logical node—much cheaper than restarting the entire logical node on failure.

One unique advantage with this approach is that checkpoints need only be taken at the context level, which provides similar advantages as dirty memory page tracking, but in a simpler design.

The second tricky piece is restarting aborted events. We note, first, that aborting events is safe because all externally-visible effects are delayed until commit time. We then note that all messages are routed through the head node before triggering events. If the head node logs the pertinent information of each message as it comes in, then uncommitted events can be trivially "replayed" upon node restart.

While the above protocol provides fault-tolerance for most physical nodes, we note that the head node is responsible for the necessary coordination; if it fails, the logical node will fail. However, because the head node looks like the entire logical node to the rest of the system, any application- or logical-node-level fault tolerance approaches should work as in the inelastic case. Section 5 discusses one approach to tolerating head-node failures using logical-node-based fault-tolerance.

5 Discussion

Preserving Failure Characteristics As we discuss above, the goal of our system is to provide transparent elasticity. In particular, we would like the application's *failure characteristics* to be similar. While this is hard to define, at a high level, we would like logical nodes to fail at the same rate regardless of how many physical nodes they are distributed over.

Beyond preventing physical-node failures from triggering logical-node failures (as the protocol described in § 4 accomplishes) there are other considerations. If the failure rate of physical nodes is sufficiently high, the additional throughput afforded by elasticity will be offset by the additional time spent recovering from faults Note that while the failure of a physical node will not cause the logical node to fail, no events that need context c will be able to make progress until the physical node is restored. This means that the run-time must "turn down" the elasticity of a logical node if physical node failure rates are reducing throughput too much.

Integration with fault tolerance at the head node In the basic programming model, no specific reliability or dependability protocol is imposed on the developer, allowing them flexibility to handle faults as they desire. However, we will optionally support compiler-automated integration of the Ken reliability protocol [16] to mask crash-restart failures. As previously mentioned, MaceKen is a Mace extension integrating the Ken reliability protocol for inelastic programming to combine event state checkpointing and message logging to mask

[3] Our failure model for now is simple fail-stop faults.

crash-restart failures. MaceKen is a per-process technique, but will not work natively with our programming model, as we use blocking RPCs for intra-node context calls, and Ken requires fire-and-forget messaging. Instead, failure masking will be provided at the *logical* node level, meaning message logging will be done at the head nodes, and event state checkpointing will be handled as already necessary to support transaction rollback on physical node failure. If non-head nodes fail, this will be masked transparently from the perspective of external observers, so no fault tolerance will be utilized. However, if the head-node fails, this is analogous to the logical node failing, and thus be handled using the Ken protocol. Specifically, when the head node restarts, the entire logical node will be rolled back to the last committed event, and event processing will continue as normal. The Ken protocol guarantees that external observers cannot distinguish a restarted node from a slowly-responding node—unacknowledged messages will just be retransmitted until they are acknowledged, while the application is free to use availability techniques to mitigate the delays caused by the apparently slow logical node.

6 Conclusions

We described a programming model that allows programmers to write applications with simple, atomic-event semantics for a fixed number of logical nodes while providing elastic execution on arbitrary numbers of physical nodes. Because elasticity raises a number of fault tolerance questions, we described a fault-tolerance protocol that would allow elastic programs to transparently mask faults and therefore fully preserve the illusion of running a program in an inelastic manner on a fixed number of nodes. We also explained how the existing mechanisms that support the elastic run-time could be repurposed to implement our fault tolerance protocol.

References

[1] BOUTEILLER, A., CAPPELLO, F., HERAULT, T., KRAWEZIK, G., LEMARINIER, P., AND MAGNIETTE, F. Mpich-v2: a fault tolerant mpi for volatile nodes based on pessimistic sender based message logging. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing* (New York, NY, USA, 2003), SC '03, ACM, pp. 25–.

[2] BRONEVETSKY, G., MARQUES, D., PINGALI, K., AND STODGHILL, P. Automated application-level checkpointing of mpi programs. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming* (New York, NY, USA, 2003), PPoPP '03, ACM, pp. 84–94.

[3] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6* (Berkeley, CA, USA, 2004), USENIX Association, pp. 10–10.

[4] GARDNER, M. Mathematical games - the fantastic combinations of John Conway's new solitaire game "life". *Scientific American 223*, 10 (October 1970), 120–123.

[5] GUO, H., WU, M., ZHOU, L., HU, G., YANG, J., AND ZHANG, L. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (New York, NY, USA, 2011), SOSP '11, ACM, pp. 265–278.

[6] HARGROVE, P. H., AND DUELL, J. C. Berkeley lab checkpoint/restart (BLCR) for linux clusters. In *Proceedings of SciDAC* (June 2006).

[7] KICIMAN, E., LIVSHITS, B., MUSUVATHI, M., AND WEBB, K. Fluxo: a system for internet service programming by non-expert developers. In *Proceedings of the 1st ACM symposium on Cloud computing* (2010), ACM, pp. 107–118.

[8] KILLIAN, C., ANDERSON, J. W., JHALA, R., AND VAHDAT, A. Life, death, and the critical transition: Detecting liveness bugs in systems code. In *NSDI* (2007).

[9] KILLIAN, C., NAGARAJ, K., PERVEZ, S., BRAUD, R., ANDERSON, J. W., AND JHALA, R. Finding latent performance bugs in systems implementations. In *FSE '10: Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering* (New York, NY, USA, 2010), ACM, pp. 17–26.

[10] KILLIAN, C. E., ANDERSON, J. W., BRAUD, R., JHALA, R., AND VAHDAT, A. M. Mace: language support for building distributed systems. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2007), ACM, pp. 179–188.

[11] LEE, H., SEIBERT, J., KILLIAN, C., AND NITA-ROTARU, C. Gatling: Automatic attack discovery in large-scale distributed systems. In *19th Annual Network & Distributed System Security Symposium (NDSS 2012)* (February 2012).

[12] LITZKOW, M., LIVNY, M., AND MUTKA, M. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems* (June 1988).

[13] LITZKOW, M., TANNENBAUM, T., BASNEY, J., AND LIVNY, M. Checkpoint and migration of UNIX processes in the Condor distributed processing system. Tech. Rep. UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[14] PEASE, M., SHOSTAK, R., AND LAMPORT, L. Reaching agreement in the presence of faults. *J. ACM 27*, 2 (Apr. 1980), 228–234.

[15] STANOJEVIC, M., MAHAJAN, R., MILLSTEIN, T., AND MUSUVATHI, M. Can you fool me? towards automatically checking protocol gullibility. In *HotNets* (2008).

[16] YOO, S., KILLIAN, C., KELLY, T., CHO, H. K., AND PLITE, S. Composable reliability for asynchronous systems. In *USENIX Annual Technical Conference (ATC)* (Jun. 2012).

[17] YOO, S., LEE, H., KILLIAN, C., AND KULKARNI, M. Incontext: simple parallelism for distributed applications. In *Proceedings of the 20th international symposium on High performance distributed computing* (New York, NY, USA, 2011), HPDC '11, ACM, pp. 97–108.