PLASMA: Programmable Elasticity for Stateful Cloud Computing Applications

Bo Sang
Purdue University, Ant Financial Services Group USA

Pierre-Louis Roman
Università della Svizzera italiana

Patrick Eugster
Università della Svizzera italiana, Purdue University, TU Darmstadt

Hui Lu
Binghamton University

Srivatsan Ravi
University of Southern California

Gustavo Petri
ARM Research

Abstract

Developers are always on the lookout for simple solutions to manage their applications on cloud platforms. Major cloud providers have already been offering automatic elasticity management solutions (e.g., AWS Lambda, Azure durable function) to users. However, many cloud applications are *stateful* — while executing, functions need to share their state with others. Providing elasticity for such stateful functions is much more challenging, as a deployment/elasticity decision for a stateful entity can strongly affect others in ways which are hard to predict without any application knowledge. Existing solutions either only support stateless applications (e.g., AWS Lambda) or only provide limited elasticity management (e.g., Azure durable function) to stateful applications.

PLASMA (Programmable Elasticity for Stateful Cloud Computing Applications) is a programming framework for elastic stateful cloud applications. It includes (1) an elasticity programming language as a second "level" of programming (complementing the main application programming language) for describing elasticity behavior, and (2) a novel semantics-aware elasticity management runtime that tracks program execution and acts upon application features as suggested by elasticity behavior. We have implemented 10+ applications with PLASMA. Extensive evaluation on Amazon AWS shows that PLASMA significantly improves their efficiency, e.g., achieving same performance as a vanilla setup with 25% fewer resources, or improving performance by 40% compared to the default setup.

Work supported by NSF grant #1618923 ("Elastic and Robust Cloud Programming"), NSF grant #1909877 ("Salvaging Commodity Operating Systems to Support Emerging Networking Technologies"), ERC grant #617805 ("LiveSoft"), DFG center #1053 ("MAKI"), and AWS Credits in Research.

EuroSys '20, April 27–30, 2020, Heraklion, Greece
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6882-7/20/04…$15.00
https://doi.org/10.1145/3342195.3387553

ACM Reference Format:
Bo Sang, Pierre-Louis Roman, Patrick Eugster, Hui Lu, Srivatsan Ravi, and Gustavo Petri. 2020. PLASMA: Programmable Elasticity for Stateful Cloud Computing Applications. In *Fifteenth European Conference on Computer Systems (EuroSys '20), April 27–30, 2020, Heraklion, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3342195.3387553

1 Introduction

Elasticity is essential to the "pay-as-you-go" cloud computing model [32], allowing cloud applications to *automatically* scale their demand for cloud resources in/out in adaptation to workload changes. Elasticity maximizes the use of resources and thus reduces infrastructure costs, meanwhile maintaining performance and service quality of cloud applications. Developers can program elastic cloud applications as a set of functions executing independently in response to specific events (e.g., AWS Lambda and Azure Durable Function). Such functions, usually encapsulated in VMs/containers, can be automatically scaled in/out on corresponding platforms [11, 22, 23, 39, 45], freeing developers and administrators from server management.

This kind of solution provides developers with ideal automatic elasticity management. However, existing automatic elasticity management provide better support to applications consisting of stateless functions, such as routines for image processing [19] or handlers of IoT devices [15]. These provide a pure transformation from input to output without external dependencies at execution. When the state of functions is thus limited to *internal* state, automating elasticity is relatively straightforward; it can simply focus on placing a function on a server node with available resources, or adjusting the number of function instantiations.

However, many cloud applications are *stateful*, i.e., functions need to share state with each other. Such scenarios are common across multiple abstraction levels, e.g., metadata of distributed file systems (one component of an application), data access tier of web applications (an entire tier or layer), microservice applications [33] (multiple loosely coupled components), and massively multi-user online games (an entire application). Those stateful applications can not be executed efficiently in state-of-the-art serverless computing platforms (e.g., AWS Lambda [4]).

Providing elasticity for generic stateful functions – or more generally components or (micro-)services – is very challenging, as an elasticity decision for one stateful component depends on not only that component, but also on others and its interaction with them. Existing programming models and frameworks enabling automated elasticity [4, 14, 23] can not capture such stateful scenarios, thus limiting the scope of the elasticity paradigm. Automating elasticity in such cases is difficult without any knowledge of applications; many non-functional requirements are hard if not impossible to learn just by looking at executions of applications. For instance one may be tempted to straightforwardly rate low-frequency component interactions as secondary to others and thus spread corresponding components across servers. Even if frequency were straightforwardly correlated with "importance", such a placement policy could adversely affect frequent interactions — of such infrequently interacting components — with others. Existing profiling approaches [2, 20] tracking system-level performance (e.g., server usage) can not connect low-level performance data to application semantics and trigger appropriate elasticity decisions.

We present PLASMA (Programmable Elasticity for Stateful Cloud Computing Applications), a novel framework for implementing expressive elastic cloud applications. It extends an existing *actor-based* [26] *application programming language* along two dimensions:

Elasticity programming language (EPL). PLASMA adds a second "level" of programming to the underlying application programming language. That is, while actors support building stateful cloud applications that have horizontal, scalable relations between stateful components [35, 53, 61], PLASMA includes a complementary *elasticity programming language* (EPL). The EPL allows users to express desired *elasticity behavior* through simple rules based on *high-level* application semantics exposed to the runtime to help it carry out fine-grained elasticity management. This is realized without violating application invariants induced by the actor programming model. In this paper we use EPL implementations for an actor-based language for building stateful distributed applications, AEON [61], but our concepts are applicable to others like Microsoft's Orleans [28] and Scala [56].

Elasticity management runtime (EMR). To guide PLASMA applications running on cloud platforms in achieving elasticity, PLASMA involves a novel *elasticity management runtime* (EMR) with two main components. (1) The *elasticity profiling runtime* tracks the behavior of actors (e.g., location, resource usage) and their interactions (e.g., message rates), as per the stated EPL elasticity policy. The information is used in making global elasticity decisions (e.g., co-locating highly interactive actors, (de-)provisioning resources) that are acted upon by (2) PLASMA's *elasticity execution runtime* leveraging a two-level architecture to reconcile global optimization accuracy with scalability.

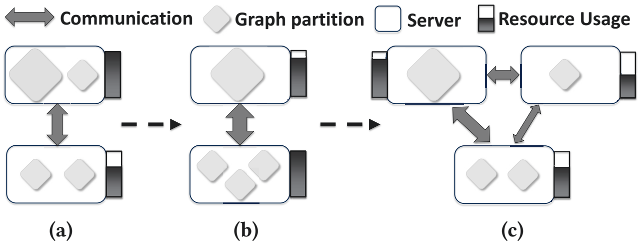


Figure 1. PageRank elasticity management example: **(a)** The initial placement of graph partitions overloads the top server which calls for partition migration. **(b)** Once migration is performed, the bottom server becomes congested. **(c)** With both servers reaching their maximum capacity, PLASMA migrates to a new server to split the load of the bottom one.

We know of no other programming framework supporting programmable elasticity for *stateful* cloud applications. Since our above-mentioned novel concepts are independent from the exact underlying actor language, we focus on those concepts in this paper. We have implemented various (10+) distributed cloud applications with PLASMA[1] including Metadata Server, E-Store [64], PageRank [31], Halo's Presence Service [51], and a Media (micro-)Service [40]. We evaluated PLASMA using these applications, showing how PLASMA enables fine-grained elasticity with only high-level user input, even outperforming application-specific elasticity solutions like Mizan [25].

Roadmap. § 2 discusses challenges in providing elasticity for stateful applications and overviews PLASMA. § 3 and § 4 detail PLASMA's EPL and EMR respectively. § 5 presents empirical evaluation. § 6 presents related work. § 7 concludes.

2 Motivation and Overview

We first motivate our work by a simple example, and further provide an overview of PLASMA in this section.

2.1 Elastic PageRank

While elasticity in cloud computing enables an efficient use of resources, no existing framework supports fine-grained elasticity management for *stateful* applications. Consequently, a whole range of essential algorithms and applications are being left behind, forcing their developers to resort to an infrastructure with suboptimal resource consumption, or build custom-tailored elastic solutions (e.g., [59, 63, 64]). PageRank [31] is a fitting and simple example of a popular stateful application. A common approach to speed up PageRank is to partition the graph it runs on into independent subsets, and have subgraphs processed in parallel. However, as partitions need to communicate with one another, deploying an elastic partitioned PageRank on a stateless platform like AWS Lambda [11] requires the use of latency-costly external storage (e.g., S3 [5], DynamoDB [37]). We have observed

[1] https://aeonproject.github.io/plasma/webpages/

25 ms average latency for DynamoDB write requests and more than 70 s to write graph vertices, edges, and partitions from a small 22 MB graph into a DynamoDB table; hence it is currently impractical to develop stateful applications requiring frequent state load/store (e.g., the distributed PageRank in § 5.4 needs to update ≈1.2 GB data at each round).

Another approach to obtain an elastic PageRank could be to directly implement the algorithm using an elastic programming language such as Orleans [53], AEON [61] or Akka [1]. Orleans balances workload by equalizing the number of actors on each server and by replicating stateless actors as the workload increases. Orleans also co-locates actors that frequently communicate with one another on the same server to avoid remote communication. AEON also evenly distributes actors across a cluster. Akka allows programmers to define router actors that forward received messages to replicated routee actors in a certain pattern (e.g., round-robin).

However, none of these languages consider server metrics (e.g., CPU usage) for ongoing elasticity management. They can not therefore properly handle applications such as PageRank – balanced graph partitioning being a notoriously difficult task [6, 18, 50, 60]. Consider the example given in Fig. 1 that provides an intuition of elasticity management for PageRank applications. In this example, a graph is split into four partitions. PageRank requires both network (i.e., partitions need to exchange data) and CPU (i.e., processing graph partitions) resources. While exact performance characteristics depend on exact graph partitions, cloud platform, and implementation, simple tests on AWS show that PageRank can be easily CPU-bound (more details in § 5.4). Assume partitions are originally evenly distributed across two servers, as in Orleans, while trying to minimize remote communication between actors as a secondary objective. But despite an even split and fair initial placement, a partition can eventually require much more computation time (Fig. 1a) to the point where the CPU consumption upper-bound of the server hosting it is crossed. With PLASMA, a developer can set CPU consumption bounds (e.g., $60\% \leq load \leq 80\%$) to ensure that a server is neither over- nor underloaded. To alleviate the load of a server, PLASMA then migrates actors to another server to respect the aforementioned CPU consumption bounds (Fig. 1b). While the migration was sufficient at first, the bottom server becomes the congested one as workloads vary, and with no available server to host additional workload, PLASMA has no choice but to spawn a new server and migrate actors to that new server (Fig. 1c).

PageRank demonstrates the need for application insights (e.g., CPU is more important than network in PageRank) for efficient elasticity management, and for platform metrics (i.e., CPU usage). As we shall show, with custom-fitted elasticity rules, PLASMA can optimize performance of various applications and adjust applications' resources to avoid under- and overprovisioning. We explore several further applications benefiting from elasticity management in § 3.3.

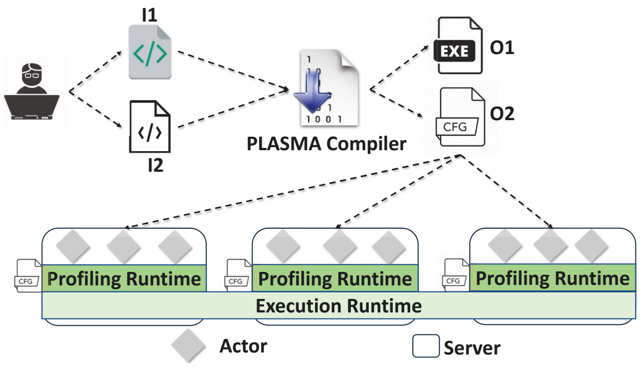


Figure 2. PLASMA toolchain overview.

2.2 PLASMA Overview

We outline how to develop elastic stateful applications with PLASMA, which is designed to extend the popular actor model [16, 26, 44]. Several corresponding languages have been already conceived for building scalable stateful elastic cloud applications (e.g., [1, 28, 35, 61]).

Programming with PLASMA. Elasticity in PLASMA is programmed *separately* from the application logic. PLASMA assumes only basic concepts for the underlying actor-based programming language (cf. Fig. 3.I); neither the underlying actor language nor the programs need to be changed to benefit from PLASMA's elasticity model. This allows programmers to easily enable elasticity for existing applications without changes, simply by defining elasticity rules in a separate program following PLASMA's *elasticity programming language* (EPL).

As shown in Fig. 2, programmers develop the (I1) application program using the actor programming language and the (I2) elasticity program using PLASMA's EPL. PLASMA's compiler takes these two programs as input and generates two output files: (O1) an executable containing both the application's binary code and PLASMA's elasticity management runtime code, and (O2) a file with elasticity actions for this application. To deploy the application programmers only need to launch the executable that takes O2 as input.

Elastic execution with PLASMA. As shown in Fig. 2, when the application is executing on a cloud platform, the PLASMA elasticity management runtime puts the EPL elasticity rules to work based on the actual performance features observed at servers about actors and their interaction. More specifically, PLASMA's *elasticity management runtime* (EMR) involves an *elasticity profiling runtime* (EPR) and an *elasticity execution runtime* (EER). The EPR keeps track of the performance of actors, focusing on those that are affected by elasticity decisions, as per the elasticity rules. Given the declared EPL elasticity rules and the performance information from the EPR, the EER takes decisions for placing actors among available servers and/or adapting the number of servers, hence realizing automated application elasticity. The EMR performs adjustments when creating actors, and every *elasticity (time) period* (set by users). Note that the EMR will not blindly follow rules to conduct elasticity management, but rather will consider the actual runtime situation (e.g., resource limitations, migration overhead) in its decisions.

The EMR does not interfere with the execution of the original language's runtime; the EPR only collects runtime data of actors. In particular PLASMA inherits the fault tolerance mechanism from the original runtime and relies on it to handle failures in the application. Failures in the EMR are handled by a separate mechanism (presented in § 4).

3 Elasticity Programming Language (EPL)

This section describes how PLASMA's EPL captures elasticity behaviors based on high-level application semantics.

3.1 Actor-based Elasticity

Given a distributed actor-based application, elasticity decisions boil down to placing/migrating actors among available servers for adjusting to workload and variations therein.

Execution features. Numerous *features* of an actor-based application's execution can be used as cues for its performance, and to drive actor placement. The features supported by PLASMA pertain to three categories:

[F-RA] Resource usage of actors (e.g., CPU).
[F-RS] Resource usage of servers (e.g., network).
[F-IA] Interaction between actors (e.g., message rate).

Elasticity rules. Similarly to the above classes of runtime features, we can classify *rules* guiding elasticity decisions based on the above features by the type of *reaction* (*behavior*) they induce on the application execution:

[R-R] *Resource elasticity rules* correspond to resource features, and strive for a better resource usage. Server resources are not directly influenced, but rather affected indirectly by adjusting the placement (and thus resource usage) of actors, and so this category of rules corresponds to both [F-RA] and [F-RS]. These rules provide programmers with a way to reserve certain amounts of resources for actors or balancing resource usage among servers. For example, programmers can specify upper and lower bounds on CPU resources for servers. If a server's CPU utilization hits the upper bound, a select group of its actors will be migrated to other servers with idle CPU resources.

[R-I] *Interaction elasticity rules* correspond to actor interaction features [F-IA]. These rules allow programmers to expose high-level application semantics to the runtime, allowing it, e.g., to co-locate actors that strongly interact, as per application semantics and actually observed at execution, thus reducing communication latency.

3.2 Syntax

Next, we detail the syntax and usage of PLASMA's EPL, realizing the above elasticity programming model. We opted for a declarative language over an imperative one as we feel that it is more natural for developers to express "policies" that way (cf. [43]). The EPL assumes only basic features of the underlying actor programming language, as shown in Fig. 3.I: a program includes a set of actors of different types (*aclass*), each declaring a set of functions (*func*) — which give rise to messages — and fields (properties – *prop*). $\overline{x}$ denotes several instances of x.

I. Actor-based application programming language basics

Program	*prog*	::=	$\overline{aclass}$	
Actor class	*aclass*	::=	$aname\{\overline{prop}\ \overline{func}\}$	
Property	*prop*	::=	*type pname*;	
Function	*func*	::=	$type\ fname(\overline{type\ \ldots})\{\ldots\}$	

II. Elasticity programming language

Policy	*pol*	::=	$\overline{rul}$	
Rule	*rul*	::=	$cond \Rightarrow \overline{beh}$;	
Actor	*actor*	::=	*atype*(*var*) | *atype* | *var*	
Actor type	*atype*	::=	*aname* | **any**	
Condition	*cond*	::=	*cond* **or** *cond* | *cond* **and** *cond*	
		|	**true** | *feat.stat comp val*	
		|	*actor* **in** **ref**(*actor.pname*)	[F-IA]
Feature	*feat*	::=	*entity.res*	
		|	*cllr*.**call**(*actor.fname*)	[F-IA]
Entity	*entity*	::=	*actor*	[F-RA]
		|	**server**	[F-RS]
Caller	*cllr*	::=	**client** | *actor*	
Statistic	*stat*	::=	**count** | **size** | **perc**	
Resource	*res*	::=	**cpu** | **mem** | **net**	
Comparison	*comp*	::=	< | > | >= | <=	
Behavior	*beh*	::=	**balance**($\{\overline{atype}\}$, *res*)	[R-R]
		|	**reserve**(*actor*, *res*)	[R-R]
		|	**colocate**(*actor*, *actor*)	[R-I]
		|	**separate**(*actor*, *actor*)	[R-I]
		|	**pin**(*actor*)	[R-I]
Value			$val \in \mathbb{N} \cup \mathbb{R}$	

Figure 3. Basic definitions for actor programming language and abstract syntax of PLASMA's EPL.

Actor-condition-behavior. The EPL (see Fig. 3.II) is used to describe – *separately* from the main program – a policy *pol* that consists of a set of elasticity rules *rul*. PLASMA's elasticity rules (both [R-R] and [R-I]) are expressed in an *actor-condition-behavior* style. That is, rules usually purport to features regarding certain *actors*, and when certain *conditions* on those features are met which may adversely affect performance, the runtime is advised to apply elasticity *behaviors* (to certain actors).

Actors. As actors of the same type tend to have similar behavior patterns, elasticity rules are expressed a priori for actor *types*, and, as detailed shortly, behaviors are expressed similarly with respect to actors of given types. A subject type of *actor* is specified by the name of the actor type's name *aname* as defined in the main application program. However, a rule can contain several actors of the same type. In order to disambiguate yet keep definitions concise by avoiding verbose variable declarations in front of every rule, we use a form of implicit variable declaration, where the use of an actor type in a rule can declare a variable *var* in an inline fashion. E.g., a condition relating to `Innernode(i)` specifies actor type `Innernode` and introduces variable `i` to refer later to all `Innernode` instances to which the condition applies.

PLASMA also introduces the special type **`any`**, allowing rules to be defined for all actors in an application. Note that PLASMA currently treats actor subtypes in the application program as distinct types from their parent types.

Conditions. Conditions are used to specify situations that may affect the performance of applications. Though rules – and thus a priori also conditions – relate to actor types, as alluded to above, conditions end up selecting a subset of actors of such a type. While actors of the same type follow the same execution logic, their actual runtime behaviors will also be affected by workloads and thus differ.

Conditions can be composed (**`and`**, **`or`**) of more elementary conditions. Basic conditions can be trivial (**`true`**) or compare statistics *stat* (highlighted in **orange**) for a runtime feature *feat* to some value *val* (a lower or upper bound), where statistics can be a number of instances (**`count`**), a **`size`** value, or a **`perc`**entage. Note that not all statistics apply to all features.

Actual features are of three basic categories (the actual features in the syntax are highlighted in **green**).

The first category (*i*) consists in conditions of the shape *actor* **`in`** **`ref`**(*actor′.pname*) which essentially select actors of the former type *actor* that are referenced by specific fields *pname* of actors of the second type *actor′*.

The second category (*ii*) corresponds to resource features of specific entities (*entity.res*). Two subcategories arise from the two types of entities considered: *actors* (*ii.a*) or **`server`**s (*ii.b*). They correspond to [F-RA] and [F-RS] respectively. The resources considered here, in turn, are of three types (**blue**): **`cpu`**, **`mem`**ory, and **`net`**work usage.

The third category (*iii*) corresponds to interaction features [F-IA] just like conditions on referencing (*i*). For specific types of messages *fname* sent from either **`client`**s or *actors* of one type to actors of another type (*cllr*.**`call`**(*actor*.*fname*)) we consider the number of such messages sent per time unit, e.g., 1 min, their size, or the percentage of a particular type of calls received by an actor, out of the total number of this type of calls received by all actors on the same server.

Behaviors. Finally, (elasticity) behaviors *beh* (**red**) tell the runtime how to react to specified conditions on given actors. There are five kinds of elementary behaviors (Fig. 3). The first two give rise to resource elasticity rules [R-R] and the others yield interaction elasticity rules [R-I].

In the former category we have **`balance`** and **`reserve`**. **`balance`**($\{\overline{atype}\}$, *res*) prompts the runtime to balance the workload on each server by migrating actors of types indicated in $\overline{atype}$ from overloaded servers to ones with idle resources. *res* refers to the type of critical resource that should be taken into consideration when balancing workloads. Note that **`balance`** does not allow type variables to be used in its first argument – using *atype* as opposed to *actor*. This is because **`balance`** targets servers instead of actors. Referring to specific actors here, programmers could add conditions on those actors (e.g., **`cpu.perc`**`>30`); then the runtime could only migrate those actors to balance the workload. This would eliminate most flexibility for the runtime (e.g., migrating actors with CPU below 30% might alleviate a bottleneck). **`reserve`**(*actor*, *res*) instructs the runtime to keep those *actors* on *dedicated* servers exclusively, whose resources are sufficient to meet the actors' demands. *res* specifies the type of desired resources on the dedicated servers.

The second behavior category spans **`colocate`**, **`separate`** and **`pin`**. **`colocate`**(*actor*, *actor*) tells the runtime to keep the concerned actors on the same server. Notice that conditions in the rule can also (further) constrain the interaction between the actors. Consider $actor_2$.**`call`**($actor_1.fname_1$).**`count`** with $fname_1$ a function of $actor_1$. Placing a condition on this term can, for instance, restrict the total number of messages $fname_1$ to $actor_1$ called by $actor_2$. Conversely, behavior **`separate`**(*actor*, *actor*) instructs the runtime to keep the actors of the two types separated whenever resources are available whilst the accompanying conditions are satisfied. This can be used for example when actors of the two types run computationally demanding activities (i.e., instead, **`colocate`** may obstruct their operations). Lastly, **`pin`**(*actor*) indicates that particular actors should not be moved. This prevents migration from hampering highly available services.

3.3 Examples

We show the use of PLASMA's EPL via five concrete examples. These applications are evaluated empirically in § 5.

Metadata Server. The Metadata Server is composed of folders and files, handled by `Folder` actors and `File` actors respectively, all of which can be opened and accessed by remote clients. Some folders are in much higher demand than others, thus receiving a significant portion of the overall number of requests. To avoid congestioning servers, we opt to migrate highly demanded folders to idle servers.

Performing such elasticity managements requires the runtime to have knowledge of application semantics, as is easily achieved with PLASMA. For instance, the aforementioned elasticity behavior can be expressed in PLASMA by defining the following elasticity rule: a `Folder` actor is migrated to an idle server (**`reserve`**) when (1) the current server's CPU usage exceeds 80% and (2) this folder receives more than 40% client requests among all `Folder` actors on this server. The rule also tells the runtime to place all `File` actors under this `Folder` actor on the same server (**`colocate`**).

Table 1. Applications implemented with PLASMA. We show elasticity rules and evaluation for the first five applications.

Application	LoC	Elasticity rules (and number of rules)
Metadata Server	253	**1.** Colocate Folder with Files in it (since they are accessed together)
PageRank [31]	465	**1.** Balance CPU workload
E-Store [64]	645	1. Put hot Partitions on idle servers 2. Colocate parent-child Partitions **3.** Balance CPU workload of Partitions
Media Service [40]	756	1. Balance network workload for FrontEnds 2. Provide VideoStream with enough CPU 3. Colocate linked VideoStream and UserInfo 4. Avoid migrating MovieReview 5. Balance CPU workload of ReviewChecker **6.** Colocate linked ReviewEditor and UserReview
Halo Presence Service [51]	314	1. Balance CPU workload of Routers **2.** Colocate Session with Players in it
B+ tree	1457	1. Colocate parent-child inner nodes **2.** Put leaf nodes on separate servers
Piccolo [57]	564	1. Balance CPU workload for Workers **2.** Colocate Worker and Table that Worker reads data from
zExpander [66]	506	**1.** Put leaf nodes on idle servers
Cassandra [47]	221	**1.** Put table replicas on different servers

```
server.cpu.perc > 80 and
client.call(Folder(fo).open).perc > 40 and
File(fi) in ref(fo.files) ⇒
   reserve(fo, cpu); colocate(fo, fi);
```

PageRank. As we introduced in § 2.1, we should balance the PageRank partitions according to CPU usage:

```
server.cpu.perc > 80 or server.cpu.perc < 60 ⇒
    balance({Partition}, cpu);
```

E-Store. E-Store [64] is an elastic partitioning framework for distributed OLTP DBMSs. Initially, root-level keys are range-partitioned into blocks of fixed size and co-located with descendant tuples. At runtime, the system monitors the workload on each server and avoids imbalance. When observing a server's resource usage (e.g., CPU) exceeding a high-water mark, the system selects k% partitions with high activity on the hot server and migrates them to idle servers. If inversely observing a server's resource usage being below a low-water mark, the system also redistributes the data.

It is a typical balancing problem where programmers need to define the conditions to trigger data distribution (i.e., high-water mark and low-water mark), and how to redistribute data (i.e., migrate hot data to idle servers). Furthermore, since data is organized in a tree structure, we can not solely migrate the hot partitions but also need to consider moving their descendants. We express E-Store needs with these 3 rules:

```
server.cpu.perc > 80 and
client.call(Partition(p1).read).perc > 30 ⇒
   reserve(p1, cpu);
Partition(p2) in ref(Partition(p1).children) ⇒
   colocate(p1, p2);
server.cpu.perc < 50 ⇒ balance({Partition}, cpu);
```

Media Service. The Media Service [40] is a more intricate stateful application, it provides two major functions, (1) rent and watch movies and (2) review movies, involving 8 types of interdependent actors in a *cloud microservice*.

Specifically, the `FrontEnd` actors are the entrance of the Media Service and are network-intensive. `VideoStream` actors stream movies to users and are CPU-intensive and latency-sensitive. A `UserInfo` actor contains the information of a user: when a user is watching a movie, the `VideoStream` actor keeps updating this user's watching history to the user's `UserInfo` actor. This yields the following three rules:

```
server.net.perc > 80 or server.net.perc < 60 ⇒
    balance({FrontEnd}, net);
server.cpu.perc>50 ⇒ reserve(VideoStream(v),cpu);
VideoStream(v).call(UserInfo(u).track).count > 0 ⇒
    pin(v); colocate(v, u);
```

In addition, users can read/write movie reviews via the `ReviewEditor` actors, which frequently interact with the `UserReview` actors by updating reviews on them. `MovieReview` actors, on the other hand, store a large amount of reviews by movie types (e.g., comedy), thus are memory-intensive. Finally, users' reviews will be checked by `ReviewChecker` actors before publication, and hence are CPU-intensive. Such semantics lead to the remaining three elasticity rules:

```
ReviewEditor(r).call(UserReview(u).update).count
    > 0 ⇒ pin(r); colocate(r, u);
true ⇒ pin(MovieReview(m));
server.cpu.perc > 90 or server.cpu.perc < 70 ⇒
    balance({ReviewChecker}, cpu);
```

Halo Presence Service. The Halo Presence Service [51] is a deployed actor-based system that tracks player liveness in Halo 4. Active game consoles periodically send heartbeat messages to the service. Each of these messages is first decrypted by a randomly selected `Router` actor, before it is forwarded to the related `Session` actor which finally forwards it to the corresponding `Player` actor.

A `Session` actor can only send messages to `Player` actors partaking in the session it manages, while `Player` actors can only belong to one session at a time. This isolation between `Session` actors and between the `Player` actors of different sessions can be leveraged to improve the system communication performance. For instance, remote messaging can be avoided by co-locating at runtime `Player` actors with their corresponding `Session` actor:

```
Player(p) in ref(Session(s).players) ⇒
   pin(s); colocate(p, s);
```

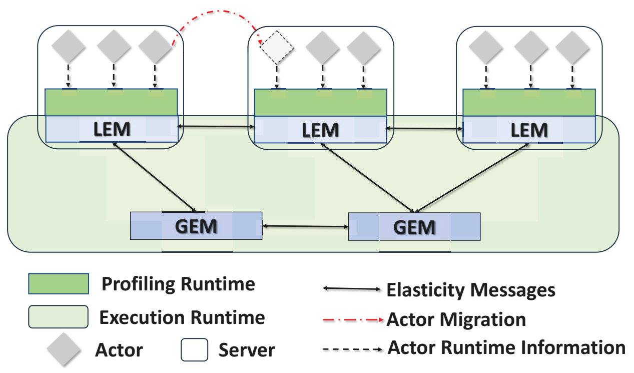


Figure 4. PLASMA's runtime system: GEMs manage application scale; LEMs handle actors of single servers.

`Router` actors require a nontrivial amount of CPU resources to decrypt messages from clients, to the point where it becomes essential to balance the CPU workload by carefully distributing `Router` actors across servers:

```
server.cpu.perc > 80 or server.cpu.perc < 60 ⇒
    balance({Router}, cpu);
```

3.4 Discussion on Language

We made several concessions for simplicity. An important choice was to consider only restricted *scopes*: in all conditions relating to interaction features , we only consider *direct* interaction. E.g., when conditioning the number of calls as in $actor_2$ `call`($actor_1.fname_1$)`.count` we only consider direct calls from an instance of $actor_2$ to $fname_1$ of one of $actor_1$, and not indirect ones, e.g., through a function $fname_3$ of intermediary actors of some type $atype_3$. Considering such transitive interaction could become quite complex: for given instances of $actor_1$ and $actor_2$, one could focus on calls through a *single* instance of $actor_3$, or any number. While both options (and more) could be expressed by introducing a larger set of variables and making existential vs universal quantification explicit, it leads to a much more complex language. Restricting scopes also simplifies profiling, especially with recursive types. § 4.3 discusses conflict resolution.

Our language is one point in the design space, and extensions such as additional features and behaviors are the subject of ongoing investigations.

4 Elasticity Management Runtime (EMR)

PLASMA's elasticity management runtime (EMR) is integrated into the runtime of the underlying actor programming language. The EMR involves an *elasticity profiling runtime* (EPR) and an *elasticity execution runtime* (EER) (cf. Fig. 4).

4.1 Elasticity Profiling Runtime (EPR)

In short, the EPR, which runs on each server, is responsible for collecting performance-related metrics, and feeding them to the EER. Corresponding to the three types of execution features of PLASMA (cf. § 3.1), the EPR collects performance information on resource usage of actors [F-RA], of the server it runs on [F-RS], and on interaction among actors [F-IA].

For [F-RS], the EPR reads system-level performance metrics from the server directly — we assume servers expose an interface such as `stat` under `/proc` in Linux. For features [F-RA] and [F-IA], the EPR keeps track of the behaviors of actors. As the elasticity rules involve specific actor types, the EPR can focus solely on the actors affected by these rules, thus limiting the overall profiling overhead. In particular, the EPR collects (1) the execution time of each task, (2) the size of actors (i.e., memory footprint), and (3) the count and size of different types of messages and involved actors. Note that the elasticity management service usually runs at every elasticity period (a configurable interval). The EPR only collects performance information since its last run.

4.2 Elasticity Execution Runtime (EER)

The EER decides on placement of actors among available servers and on adapting the number of required servers.

Two-level architecture. PLASMA's two-level EER design includes *local* and *global elasticity managers* (LEMs and GEMs respectively). This design makes a practical tradeoff between *scalability* (distributed LEMs with local views and actions) and *accuracy* (centralized GEMs to provide a more complete, global, view of the system). Duties are thus split in that a LEM works only for a single server and is responsible for interaction elasticity rules [R-I] as these can be monitored locally. On the other hand a GEM oversees a group of servers, and is responsible for resource elasticity rules [R-R] among these servers, as these need to place actors among a group of available servers, requiring global performance information.

LEMs. When an elasticity management cycle begins, a LEM reads the performance information of the server and actors from its local EPR. The LEM summarizes such information, and reports *imperative* information (e.g., focusing on actors furthest beyond thresholds) to its GEM. The LEM also iterates through its actors and checks related interaction elasticity rules [R-I] (if any). If the conditions in the rules are satisfied, the LEM identifies *executable actions*. Take for example a `colocate` rule specifying actors of two types $atype$ and $atype'$, with frequent interaction through some function $fname$, e.g., $atype$`.call(`$atype'.fname$`).count` `> 1000`. The LEM goes through each $atype$ actor and checks the message count between it and remote $atype'$ actors generating a migration action when a message count is larger than the threshold defined. Notice that the LEM managing the $atype'$ actor does the same but on a different server. The two LEMs communicate directly, without passing by a GEM, to decide whose actor to migrate to the other's server (by default the one with lower resource usage).

GEMs. A GEM manages a subset of servers; each server is managed by one GEM *at a time*. After a certain time period (e.g., elasticity time period), each LEM randomly picks a new

Table 2. API summary.

(a) GEM & LEM local functions

Function	Functionality
getActRules	Return actor elasticity rules
getActorsRuntime	Return runtime info of all local actors
applyActRules	Return migration actions as per actor runtime info and elasticity rules on LEMs
getResRules	Return resource elasticity rules
collectActorsFResRules	Return runtime info of actors related to resource elasticity rules
getServerRuntime	Return local server runtime info
resolveActions	Resolve conflicted migration actions of LEMs and GEMs. Return final actions
applyResRules	Return migration actions according to actor and server runtime info and resource elasticity rules on GEMs
checkIdleRes	Decide if one server has enough idle resources to accept an actor

(b) Action datatype

Action datatype field	Content
actor	Actor for migration
srcServ	Server currently holding the actor
trgServ	Target server for actor migration

GEM. After receiving the profiling information from LEMs of its managed servers, a GEM creates a *global runtime snapshot* for all its managed servers. Referring to this snapshot, the GEM checks the conditions of resource elasticity rules. If any are met, the GEM identifies executable actions (O2 in Fig. 2) from the rules, and tasks servers.

Take for example a `balance` rule. When its condition is met (typically a lower or upper bound on server resources is exceeded), the GEM migrates a select set of actors among its managed servers to balance workload. PLASMA uses a simple heuristic to thus select actors: a GEM only migrates actors from overloaded servers (i.e., with resource usage above upper bounds) to servers with enough idle resources – especially below specified lower bounds. If all of a GEM's managed servers are overloaded (resp. under-utilized), it broadcasts an *adjustment* message to all other GEMs. GEMs reply whether their observations are similar. If the majority of replies received by the requester GEM corroborate its own view, it increases (resp. decreases) the number of servers.

LEM-GEM interaction. Alg. 1 and Alg. 2 outline how LEMs and GEMs coordinate on generating migration actions based on elasticity rules and runtime performance information (using APIs summarized in Tab. 2). Each LEM (Alg. 1) reads actors' runtime information from the profiling runtime and identifies migration actions (line 7) based on actor elasticity rules (line 5). The LEM then checks resource elasticity rules and reports related actors' runtime information as well as its server runtime information to a GEM (line 12).

The GEM (Alg. 2) starts processing reports from LEMs when it receives enough of those (line 8). It only checks

Algorithm 1 Elasticity execution on LEM *lem*

```
 1: Local variables:
 2:   existActors          ▷ local actors and actors to be migrated to this server
 3:   gems                                                  ▷ addresses of GEMs
 4: task processElasticity do
 5:   actRules ← getActRules()
 6:   actorsRT ← getActorsRuntime()
 7:   lemActions ← applyActRules(actorsRT, actRules)
 8:   resRules ← getResRules()
 9:   ractorsRT ← collectActorsFResRules(actors, resRules)
10:   serverRT ← getServerRuntime()               ▷ collect server runtime info
11:   gem ← $gem_x$ | $gem_x$ ∈ gems                         ▷ pick random GEM
12:   send (REPORT, ractorsRT, serverRT) to gem             ▷ report to GEM
13:   wait until receive (RREPLY, gemActions) from gem
14:   finalActions ← resolveActions(lemActions, gemActions)
15:   for all (action ∈ finalActions) do
16:     send (QUERY, action) to action.trgServ             ▷ can server accept
17: upon receive (QUERY, action) from lem′ do
18:   if checkIdleRes(existActors, action.actor) then
19:     existActors ← existActors ∪ {action.actor}           ▷ take resources
20:     send (QREPLY, action) to lem′
21: upon receive (QREPLY, action) do
22:   migrate action.actor to action.trgServ
```

resource elasticity rules and generates corresponding migration actions (line 10) and informs relevant LEMs (line 14). A LEM and a GEM can generate different, potentially conflicting, actions for the same actor. A LEM then resolves such conflicts once it receives migration actions from its GEM (line 14). Finally, the LEM starts migrating actors when the target server agrees to accept them (line 22).

New actor creation. When the application creates an actor (of type *atype*) on a server, this server's LEM queries the GEM, which managed it during last the elasticity period, where to place the new actor. The GEM checks relevant elasticity rules and decides the initial placement. E.g., the rules require to co-locate *atype* actors with references, or identify *atype* actors as CPU-intensive. Then the new actor is to be co-located with another actor that has a reference to it, or put on a server with idle CPU resources. If the GEM can not find a valid rule for *atype* actors, it randomly picks a server from the ones it manages. With the help of input elasticity rules, PLASMA has a higher chance to place new actors on the right servers from the start, as we will see shortly.

4.3 Discussion on Runtime

Conflict resolution. As stated above, LEMs and GEMs identify executable actions based on rules. These actions are enqueued at LEMs, and are executed by the actor programming language's runtime via its live actor migration procedure. However, as programmers may define multiple elasticity rules for one actor type, conflicts may arise. PLASMA provides two mechanisms to resolve these. (1) When *compiling* elasticity rules, PLASMA's compiler detects conflicting

Algorithm 2 Elasticity execution on GEM *gem*

```
1:  Local variables:
2:     $actorsRTs$          ▷ queue for actor runtime information from LEMs
3:     $serversRTs$         ▷ queue for server runtime information from LEMs
4:     $actionQs$           ▷ map of addresses to action queues
5:  upon receive (REPORT, $actorsRT$, $serverRT$) from $lem$ do
6:     $actorsRTs \leftarrow actorsRTs \oplus \{actorsRT\}$
7:     $serversRTs \leftarrow serversRTs \oplus \{serverRT\}$
8:     if $|servers| > K$ then                ▷ $K$ is given by user
9:        $resRules \leftarrow$ getResRules()
10:       $actions \leftarrow$ applyResRules($actorsRTs$, $serversRTs$, $resRules$)
11:       for all ($act \in actions$) do
12:          $actionQs[act.\mathrm{srcServ}] \leftarrow actionQs[act.\mathrm{srcServ}] \oplus \{act\}$
13:       for all ($addr \mid \exists\ actionQs[addr]$) do
14:          send (RREPLY, $actionQs[addr]$) to $addr$     ▷ ret to LEM
```

rules for the same actor type, and issues warnings. (2) When applications are *running*, LEMs resolve the remaining conflicts by choosing the migration action for an actor with the highest priority, which can be specified by programmers or determined by assigning priorities to migration actions. E.g., `colocate` can break the resource elasticity actions of `balance` in that multiple LEMs might try to migrate their actors to the same server and overload it. If PLASMA prioritizes `balance` over `colocate`, it will only allow the target server to accept actors if it has enough resources. Existing conflict resolution approaches [30, 42, 47] can also be leveraged in PLASMA; they are beyond the scope of this paper.

Fault tolerance. As is evident from Alg. 1 and Alg. 2, no state synchronization is required between LEMs and GEMs or among GEMs. Hence, if a GEM fails while computing the set of migration actions, LEMs can still progress by picking another GEM through the shuffling process described. We run multiple GEMs for scalability and fault tolerance when evaluating PLASMA in § 5.

Actor placement stability. We opt for a conservative policy to actor migration to minimize the cost associated with actor state "re"-migrations. More aggressive migration policies [46] could be employed, e.g., by pre-profiling actor resource consumption or migrating more actors than strictly needed, but no optimal policy exists.

To avoid frequent actor migrations, an actor can only be migrated if it stayed on the same server for a certain time, which is set to be equal to the elasticity period (cf. § 2.2).

5 Evaluation

The concepts of PLASMA can be implemented in many actor programming languages.

5.1 Synopsis

We evaluate our approach through an implementation in AEON [61] involving 3500 Python LoC added to the AEON compiler, that parses both PLASMA elasticity rules and AEON program to generate an elasticity configuration file (PLASMA compiler in Fig. 2). We also extend AEON's runtime by 5000 C++ LoC to collect actors' and platform's runtime information (profiling runtime in Fig. 2), and conduct elasticity management (execution runtime in Fig. 2). We chose AEON over Orleans [53] and Akka [1], as when starting our prototyping, Orleans' code-base was undergoing frequent significant updates while Akka lacks live actor migration features.

We evaluate PLASMA with several stateful applications on Amazon AWS. The elasticity rules used for each scenario are described in § 3.3, with their summary in Tab. 1, demonstrating the low effort with which a multi-actor application can be complemented with PLASMA.

We first evaluate the overhead of PLASMA's runtime (§ 5.2). Next we demonstrate how a simple elasticity rule leveraging application-specific knowledge improves a Metadata Server's elasticity management (§ 5.3). We compare the efficiency of PLASMA against the state of the art on an elastic PageRank (§ 5.4). Then we showcase how PLASMA can help developers implement specific elasticity management in E-Store (§ 5.5). We show how PLASMA handles highly dynamic workloads in a Media Service (§ 5.6). Finally, we evaluate how different number of GEMs impact the performance of the Halo Presence Service (§ 5.7).

5.2 PLASMA's Runtime Overhead

First we assert that the EPR does not impose high overheads on applications when tracking performance data. The EPR only collects runtime information of actors on the server it is deployed on, the EPR overhead is therefore only affected by the number of actors and messages on a single server, regardless of the number of servers used by this application.

To this end, we use an online chat room microbenchmark where users, represented each by an actor, can exchange messages with others within the same room. The EPR tracks information on all messages (e.g., type, size, number) and the times for actors to process them. The chat room is deployed on a single AWS instance, i.e., actors are stationary, and is tested with different numbers of users. Tab. 3 shows the profiling overheads of PLASMA's EPR on the chat room actors by normalizing the execution time of PLASMA with that of a vanilla system without elasticity (e.g., 1.007 means 7‰ overhead). The setup x-instance refers to the number of users x, with $x \in \{8, 16, 32\}$, deployed on either a m1.small instance identified as s or a m1.medium instance identified as m. In all setups, users keep generating messages at high rates to put pressure on the server's CPU. In this overloaded situation we never observe more than 2.3% overhead, showing that message latency is virtually unaffected by profiling.

While the overhead of the EER is highly related to the elasticity rules, we do not observe any noticeable overhead (i.e., over 1%) on any of the applications we evaluate. The EER overhead remains low thanks to: (1) its periodical execution, the EER only executes for a couple of seconds per period in our scenarios, and (2) the low rule count (i.e., less than 10) needed to cover applications elasticity requirements.

Table 3. Normalized EPR overhead.

8-s	16-s	32-s	8-m	16-m	32-m
1.007	1.001	1.023	1.003	1.006	1.005

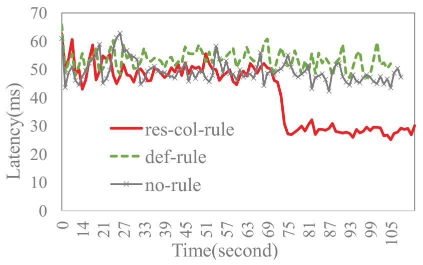


Figure 5. Simple reserve & colocate vs default vs no rule in Metadata Server.

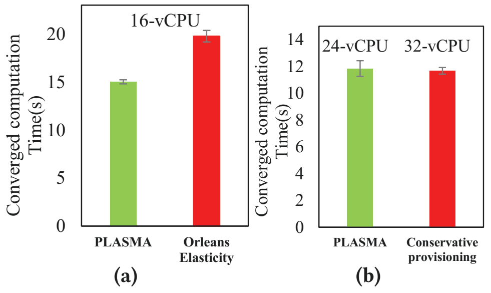


Figure 6. PageRank PLASMA's vs Orleans' elasticity, **(a)** static & **(b)** dynamic allocation.

5.3 Metadata Server

In our first scenario we display the effect of a simple mix of *Resource elasticity rules* and *Interaction elasticity rules* in PLASMA when deployed on a Metadata Server (cf. § 3.3).

In the experiment, we create 4 folders with 8 files in each. The server is deployed on an AWS m1.small instance, and 16 clients on another m1.medium instance. This setup overloads an m1.small instance, i.e., simulates a service under high demand. Among the 4 `Folder` actors, 1 actor receives 50% of requests from clients, and the other 3 evenly share the remaining 50%. `File` actors in a same folder have the same workloads. We compare three setups: (1) res-col-rule executes the **`reserve`** and **`colocate`** elasticity rule defined in § 3.3; (2) def-rule mimics a default rule, simply migrating actors with heavy workload (i.e., `Folder` actors) to an idle server; (3) no-rule does not conduct any elasticity management. The first two setups require an extra server; they use an elasticity time period of 80 s. We run each setup for ≈100 s to collect enough data before and after elasticity management.

Fig. 5a shows that the elasticity rule (res-col-rule) reduces latency by 40% compared to both other setups. The second setup (def-rule) however does not display any visible latency benefit compared to the setup without elasticity (no-rule) because accessing a folder implies accessing the files contained in it, even when the `Folder` and `File` actors are on different servers. Therefore all accesses to a `Folder` actor on one server end up being forwarded to `File` actors on another server, provoking an overheard that nullifies the potential migration gains. This demonstrates the importance of application knowledge in elasticity management.

5.4 PageRank

In this scenario, we show the efficiency of PLASMA on a distributed actor-based variant of the popular PageRank [31] algorithm. We focus on the basic algorithm without specific optimizations as these do not address workload imbalances.

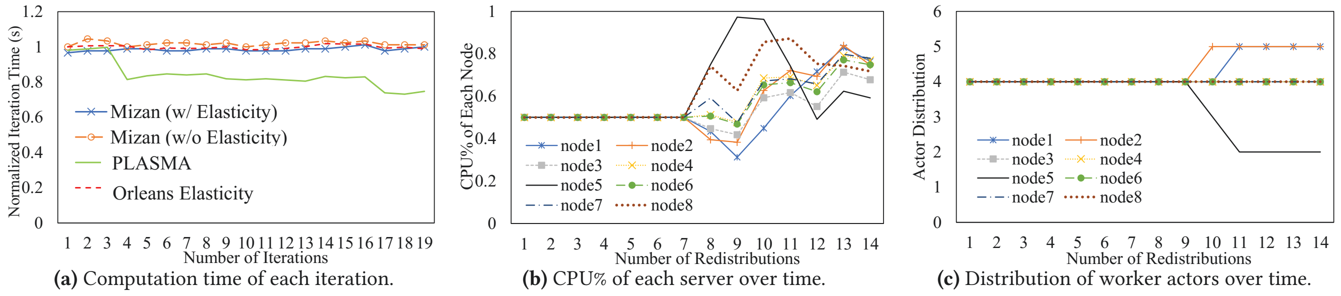
In our implementation, each `Worker` actor iteratively computes on one partition and synchronizes at the end of each iteration with the other workers to exchange computation results. Since all workers synchronize at the same time, the overall execution speed is limited by the slowest worker.

Though many partitioning schemes and systems have been proposed for partitioning graphs [6, 18, 50, 60] and thusly balancing workloads across workers, ensuring a fair workload distribution remains a non-trivial task. We use SNAP's LiveJournal online social network [24] as dataset. The graph is split with the popular graph partitioning tool METIS [18] that computes balanced partitions.

Dynamic workload balance. We first show how PLASMA balances workloads among a *fixed* set of resources. This experiment uses 8 VMs, each being an AWS m5.large instance (2 vCPUs and 8 GB memory), for a total of 16 vCPUs that are connected with 10 Gbps links. All runs are congestion-free.

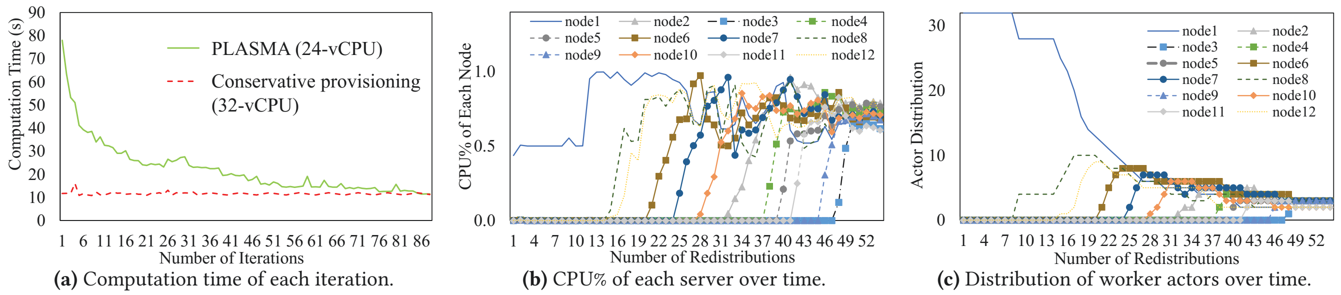
We first compare PLASMA with elasticity management to the limited elasticity management of Orleans consisting in attempting to balance workload by putting the same number of actors on each server. We implement the same elasticity rules in AEON. We use METIS to evenly split the graph in 32 partitions, resulting in 32 actors, and *randomly* assign them across the 8 VMs. Since the number of actors is already balanced across servers, Orleans elasticity management does not take further action. PLASMA uses the same partition assignment, but combines it with a **`balance`** resource elasticity rule that sets the lower bound to 60% and upper bound to 80% (as for Piccolo in § 3.2). Once the PageRank application starts, PLASMA's EMR balances the worker actors among the 8 VMs based on their CPU resource usage, while the worker actors stay in the same VMs with Orleans' elasticity. The experiment is repeated 3 times for each of the 5 different random distributions used. Fig. 6a shows that PageRank converges 24% faster with our elastic solution PLASMA redistributing actors, compared to Orleans elasticity management.

Fig. 7b and Fig. 7c show the detailed behavior in a given experiment run of the CPU consumption for each server and the per-server actor distribution respectively. Once worker actors finish reading data and start iterative computations, the CPU usage of each server starts diverging greatly despite the even partitioning performed by METIS. PLASMA detects load imbalance and moves worker actors from overloaded servers (e.g., server 5) to under-utilized ones (e.g., servers 1 and 2, until the CPU usage of servers falls between the



(a) Computation time of each iteration. (b) CPU% of each server over time. (c) Distribution of worker actors over time.

Figure 7. PageRank dynamic workload balance. PLASMA achieves 24% faster iteration times after initial automated balancing. (In **(b)** and **(c)**, each server is busy reading data in the early re-distributions.)



(a) Computation time of each iteration. (b) CPU% of each server over time. (c) Distribution of worker actors over time.

Figure 8. PageRank dynamic resource allocation. PLASMA achieves the same application-level performance with 12 servers in comparison with the conservative provisioning case using 16 servers (with one worker per vCPU).

upper and lower bounds (i.e., 60%-80%). As a result, PageRank converges faster since the computation time of each partition is more homogeneous with elastic PLASMA, and no one worker actor is lagging behind for the system-wide synchronization happening at the end of each iteration. This experiment clearly shows that under fixed resources (e.g., CPU), PLASMA can detect workload imbalance and improve resource efficiency by automatic actor re-location.

We further compare PLASMA with Mizan [48], a state-of-the-art graph processing system for dynamically balancing computation across servers via graph vertex migration. We run the open source Mizan [25] with both its default configuration (without elasticity) and its dynamic migration scheme (with elasticity) in the same setting as PLASMA– 8 AWS m5.large VMs and the LiveJournal graph with 32 partitions. Since the absolute iteration time of Mizan is about 4× longer than that of PLASMA, to compare elasticity effectiveness of the two solutions, we normalize each iteration time to the first iteration of the respective case without elasticity. Fig. 7a shows Mizan with elasticity reduces iteration time by up to 3% compared to the case without elasticity. In contrast, PLASMA with elasticity reduces iteration time by up to 24% compared to the case without elasticity, showing that PLASMA balances load more effectively while being generic.

Dynamic resource allocation. Trivially we can reach best PageRank convergence time by over-provisioning resources (conservative provisioning). We thus run 16 AWS m5.large instances for a total of 32 vCPUs, randomly mapping each of the 32 worker actors to their own vCPU. As expected, the convergence time is nearly halved (11.67 s, cf. Fig. 6b) compared to that of a 16-vCPU setup (19.77 s, cf. Fig. 6a). But can we achieve the same (or close) performance using fewer resources? To answer this question, we set PLASMA to allocate resources *dynamically* – we re-use the above `balance` rule, but once all of the existing servers are overloaded, PLASMA provisions a new server (cf. § 4.2). In our experiments (cf. Fig. 8), we start with one running server for PLASMA and place all 32 worker actors on it. Fig. 8b clearly shows PLASMA provisioning new servers (via AWS Instance Scheduler [10]) until it reaches a stable state where the CPU usage of every server is within the defined lower and upper bounds. Fig. 8c shows the details of the worker actor re-distributions as PLASMA provisions new instances. Performance improves each round as PLASMA performs elasticity management gradually (cf. § 4.3) and inches towards an optimal actor distribution. Eventually, PLASMA performance comes very close to the best performance as shown in Fig. 8a for one run and in Fig. 6b for the average across runs. PLASMA achieves nearly identical performance with 12 servers as the over-provisioning case does with 16 servers, saving 25% of resources.

5.5 E-Store

Programmers often end up implementing specific elasticity management in their applications without help from specialized tools. We show here that PLASMA can deliver similar performance as in-app implemented elasticity management.

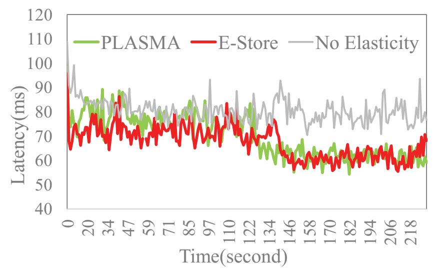
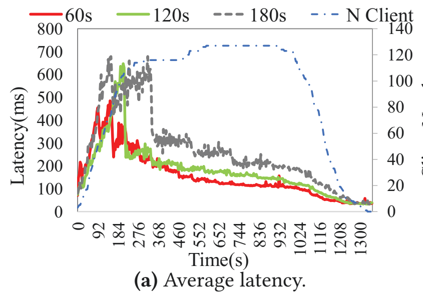
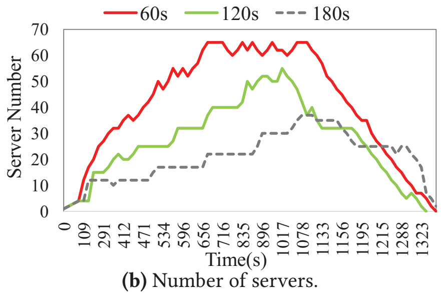


Figure 9. Latency of E-Store application. Similar for E-Store and PLASMA E-Store.



(a) Average latency.



(b) Number of servers.

Figure 10. Elasticity management for the Media Service. A small elasticity period lowers the latency and fasten resources allocation/reclaiming.

As detailed in § 3.3, E-Store [64] has to migrate hot key partitions to handle unbalanced workloads. At the same time, root level partitions must also be co-located with their descendants to avoid remote communication. We implemented E-Store in AEON [61] and added 3000 LoC to its runtime to include the specific platform details (e.g., CPU usage, actor placement) used by E-Store for elasticity. We compare the performance of this AEON E-Store with PLASMA E-Store that (only) executes the elasticity rules defined in § 3.3.

We evenly deploy 40 root level partitions of E-Store on 4 m1.small instances. Each such partition has 4 child partitions. Querying E-Store are 48 clients on another 2 m1.medium instances, generating unbalanced workload on partitions. The first root partition receives 35% of total requests; the second receives 35% of the remaining 65% of requests; the third receives 35% of the requests remaining after that, aso. Requests arriving at a root partition will continue to access one child partition randomly. We also run a non-elastic version for comparison. During execution, AEON E-Store and PLASMA E-Store both require an extra instance.

As Fig. 9 shows, the performance of AEON E-store and PLASMA E-store are close to each other, and they both show obvious performance improvement compared to AEON E-Store without elasticity management (No Elasticity). After detailed analysis, we find elasticity behaviors in both versions are quite similar despite differences in their concrete elasticity management. For example, AEON E-Store migrates the top k% root partitions on overloaded servers to idle servers while PLASMA picks root partitions that receive a certain percentage of requests among all root partitions on the same server. This demonstrates the usefulness of PLASMA for implementing application-specific elasticity behavior.

5.6 Media Service

We next show how PLASMA improves the Media Service (§ 3.3) performance with highly dynamic workloads, with a focus on showing the impact of elasticity time periods.

We deploy 128 clients on 8 AWS m1.small instances. In the first 10 minutes, these clients join the service (i.e., start making requests) following a normal distribution (μ = 2 min, σ = 90 s) and they keep sending requests for 4 more minutes. They all leave the service starting from the 14 minute mark, for a period of 10 minutes, following a normal distribution (μ = 19 min, σ = 90 s). Half of the clients' requests are "watch movie" and half are "review movie".

On the server side, the Media Service is deployed on multiple m1.small instances, 4 instances initially, and can scale up to 65 instances. One `UserInfo` or `UserReview` actor only serves one client while all other actors serve two clients each (e.g., `FrontEnd`). More actors are created as more clients access the service. As the number of clients (i.e., workload) evolves, PLASMA's runtime gradually adjusts those actors' placement to ensure low request latency. We start 1 GEM to execute elasticity rules as described in § 3.2 and run a scenario per elasticity period of 60 s, 120 s and 180 s.

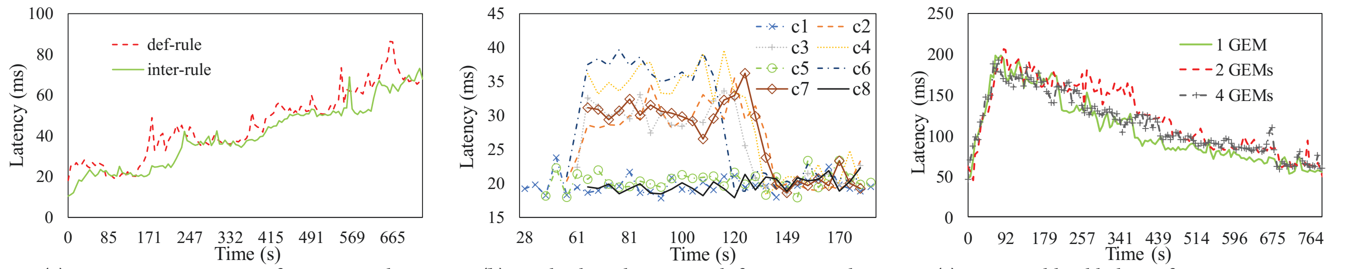
As shown in Fig. 10a, a smaller elasticity time period enables a better responsiveness from PLASMA's runtime, with the 60 s elasticity period displaying the best latency results. Fig. 10b details the number of servers used by the application over time under different elasticity time periods: PLASMA's runtime — with shorter elasticity time period — can allocate and reclaim resources in a faster manner corresponding to the workload dynamics. Yet too frequent elasticity management may incur additional overhead. This clearly shows PLASMA works well with more (e.g., 6) elasticity rules under dynamic workload, with a well-chosen elasticity time period.

5.7 Halo Presence Service

In this scenario, we show how to improve performance of the Halo Presence Service [51] with rules for different types of actors, and assess the effect of the number of GEMs used.

Interaction elasticity rule. As discussed in § 3.3, a `Session` actor can only send messages to `Player` actors in it, suggesting that they be co-located. We could also instead use a rule that co-locates actors that *frequently* interact with one another, but this rule can lead to poor migration choices (e.g., if a router actor happens to send many messages to one session actor for a while). Moreover, *frequent* can have various interpretations. We use this frequency-based rule as our default rule (baseline) for our evaluation.

The experiment setup is composed of 8 `Router` actors and 8 `Session` actors that are deployed on 8 AWS m1.small



(a) Latency: interaction vs frequency rule. **(b)** Single client latency with frequency rule. **(c)** CPU workload balance for router actors.

Figure 11. Elasticity management for Halo Presence Service. **(a)** shows the elasticity rules enable smoother player latency evolution, **(b)** shows the importance of actors colocation, and **(c)** shows the slight impact on latency of number of used GEM(s).

servers, with one of each actor type deployed per server. To highlight remote messaging latency, router actors do not perform decryption and forward messages directly. We simulate players behind game consoles by starting 32 clients on another 2 AWS m1.medium servers. The 32 clients join the game in 4 rounds (of 180 s each), with 8 clients joining per round, each at a random time during the round. A joining client is assigned to a session and the application creates a corresponding `Player` actor for it. Depending on the rule in place, this new `Player` actor either (1) gets co-located with its session actor when the aforementioned elasticity rule is established, or (2) is first placed on a random server and the default rule attempts to co-locate it with the actors it frequently interacts with. Only one GEM is started and the elasticity period is set to 70 s.

Fig. 11a depicts the average message latencies resulting from the two rules. With the elasticity rule (inter-rule), clients largely avoid remote messaging from the start of the experiment while the default rule (def-rule) leads to degraded performance for certain timespans (e.g., 0 s–85 s, 171 s–247 s). Only once `Player` actors are co-located with their `Session` actor do message latencies become similar (85 s–171 s, 247 s–332 s). The elasticity rule enables smoother latency evolution for an enhanced user (i.e., Halo player) experience.

Fig. 11b shows the detailed performance of each client for the first round of a single run under the default rule. Out of the 8 joining clients, c1, c5 and c8 are fortuitously instantiated on the right servers for their `Session` actor, while c2, c3, c4, c6 and c7 experience a latency between 30 ms and 40 ms, which is ≈35% higher than that of well-placed clients. All clients' latency is reduced to 20 ms after re-distribution (after 70 s of presence). Note that high latency in the first few minutes may limit a player's interest in the game. Attempting to obtain better results by shortening the elasticity period might lead to overzealous actor migration that can worsen performance.

Resource elasticity rule. As mentioned in § 3.3, we need to provide `Router` actors with enough CPU resources. To demonstrate the efficiency of this rule, we define a setup made of 64 `Session` actors and 32 `Router` actors and deploy them on 64 AWS m1.small servers. Each `Session` actor is hosted on a separate server whereas `Router` actors are initially evenly distributed across 8 of these servers. We run up to 128 clients (i.e., 128 `Player` actors) on 8 AWS m1.medium servers with a varying number of GEMs: 1, 2, and 4. The elasticity time period is set to 80 s.

Fig. 11c shows a sudden rise in average latency as more and more clients join the game, indicating that the 8 servers with `Router` actors are overloaded. In response, the GEM(s) start to balance the workload as per the resource elasticity rule until each `Router` actor ends on a server with enough resources, allowing latency to stabilize. Additionally, we see that deploying several GEMs, for scalability and fault tolerance, only has a small impact on latency.

6 Related Work

Stateless/state-agnostic elasticity. Infrastructure-level elasticity services for the cloud are typically provided through auto-scaling [7–9, 12]. The number of VMs etc. can be automatically adjusted based on predefined policies and metrics. To use such infrastructure-level elasticity, cloud applications must follow a *very specific programming model*. E.g., service-oriented programming [29, 55] allows each service of a web application to be scaled in/out via sharding or partitioning [34, 36]. Machine learning techniques have been used for elastic provisioning in such models based on workload change prediction [41, 54]; these however remain coarse-grained.

Serverless computing is a recent trend in elastic cloud programming [11, 22, 23, 39, 45]. It allows developers to decompose cloud applications into state*less* functions, deployed and scaled elastically. E.g., AWS Lambda [11] allows users to upload their function code to the cloud; management and capacity planning is done automatically. PLASMA extends the scope of serverless computing to *stateful* applications.

Elasticity for stateful applications. Programming stateful applications in extant serverless computing requires leveraging a storage tier (e.g., database, cross-application cache, network file/object store) to store state across functions [5, 17, 21, 57, 65, 67]. Yet relying on such a storage tier fails to exploit inherent data locality (of accessing functions) and thus limits effectiveness of application elasticity. Scaling the storage tier automatically is notoriously hard [28, 35, 53, 61].

Many approaches provide elasticity management for *specific stateful applications* [27, 38, 48, 59, 64]. E.g., ElastMan [27] includes an elasticity manager for key-value storage in multi-tier web applications. E-Store [64] and Mizan [48], introduced earlier, realize elastic partitioning for distributed OLTP databases and graph vertex migration for map-reduce based graph processing systems [6, 50, 60] respectively.
Ray [52] is a reinforcement learning framework that supports stateful computation with actors. The elasticity management of these solutions is system/application-specific and *fail to provide general solutions.*

Both Locus [58] and Pocket [49] provide a more efficient storage solution for serverless computing. Locus combines cheap but slow storage with fast but expensive storage to achieve good performance and cost-efficiency at the same time. Pocket is an elastic distributed data store for serverless computing that provides similar performance as ElastiCache Redis [3] at a lower cost. While Locus and Pocket tackle the performance of serverless computing from the storage side, PLASMA focuses on elasticity management by, for example, colocating two actors (or functions) that frequently interact.

Azure durable functions [13] allow programmers to implement stateful functions for their serverless applications. Programmers however can not customize their application's elasticity management like they can with PLASMA.

Elastic actor programming languages. The actor programming model [26] allows building applications with scalable (a prequisite for elasticity) relations between entities, well-studied in the context of cloud applications, e.g., in EventWave [35], Orleans [53] and AEON [61]. Though these languages support live actor migration, they do not provide automated elasticity management. Previously [62] we sketched the case for elasticity programming, however only providing coarse-grained monolithic constructs instead of fine-grained elasticity conditions and behaviors, and without detailed and evaluated runtime techniques for elasticity action execution.

7 Conclusions

Existing automatic elasticity solutions (e.g., serverless computing) simplify development and deployment of distributed applications executing in third-party infrastructure by providing simple abstractions such as functions, and dealing with resource provisioning completely automatically in the face of fluctuating workloads. PLASMA introduces the same benefits to state*ful* applications by complementing the actor-based programming model with: (1) a second "level" of programming for delineating *actor-condition-behavior* rules that drive elasticity management; (2) an elasticity-aware runtime that accordingly profiles actors of specified types and applies corresponding actions. While a core design goal was to keep PLASMA's elasticity programming language simple, we are investigating several extensions.

Acknowledgments

We thank our shepherd Kiran-Kumar Muniswamy-Reddy as well as the EuroSys '20 reviewers for their helpful feedback.

References

[1] Akka. https://akka.io/.
[2] Amazon CloudWatch. https://aws.amazon.com/cloudwatch/.
[3] Amazon ElastiCache Redis. https://aws.amazon.com/elasticache/redis/.
[4] Amazon Lambda Programming Model. https://docs.aws.amazon.com/lambda/latest/dg/gettingstarted-features.html#gettingstarted-features-programmingmodel.
[5] Amazon S3. https://aws.amazon.com/s3/.
[6] Apache Incubator Giraph. http://incubator.apache.org/giraph/.
[7] Autoscaling. https://aws.amazon.com/autoscaling/.
[8] Autoscaling Groups of Instances. https://cloud.google.com/compute/docs/autoscaler/.
[9] Autoscaling with Heat. https://docs.openstack.org/senlin/latest/scenarios/autoscaling_heat.html.
[10] AWS Instance Scheduler. https://aws.amazon.com/answers/infrastructure-management/instance-scheduler/.
[11] AWS Lambda. https://aws.amazon.com/lambda/.
[12] Azure Autoscale. https://azure.microsoft.com/en-us/features/autoscale/.
[13] Azure Durable Function. https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-overview.
[14] Cloud Functions Programming Model. https://docs.microsoft.com/en-us/azure/azure-functions/functions-reference.
[15] Cloud IoT Core. https://cloud.google.com/iot-core/.
[16] Erlang. https://www.erlang.org/.
[17] Memcached. http://memcached.org/.
[18] METIS Graph Partition Library. http://exoplanet.eu/catalog.php.
[19] Module: Coordinate a Serverless Image Processing Workflow with AWS Step Functions. https://github.com/aws-samples/aws-serverless-workshops/tree/master/ImageProcessing.
[20] Overview of Azure Monitor . https://docs.microsoft.com/en-us/azure/monitoring-and-diagnostics/monitoring-overview-azure-monitor.
[21] Redis. https://redis.io/.
[22] Serverless. https://cloud.google.com/serverless/.
[23] Serverless Computing. https://azure.microsoft.com/en-us/overview/serverless-computing/.
[24] SNAP. https://snap.stanford.edu/data/.
[25] The Graphs Blog. https://thegraphsblog.wordpress.com/presentations/.
[26] G. Agha and C. Hewitt. Actors: A conceptual foundation for concurrent object-oriented programming. In *Research Directions in Object-Oriented Programming*, pages 49–74. 1987.
[27] A. Al-Shishtawy and V. Vlassov. Elastman: autonomic elasticity manager for cloud-based key-value stores. In *The 22nd International Symposium on High-Performance Parallel and Distributed Computing, HPDC'13*, pages 115–116, 2013.
[28] P. A. Bernstein, S. Bykov, A. Geller, G. Kliot, and J. Thelin. Orleans : Distributed virtual actors for programmability and scalability. Technical report, Microsoft Research, 2014.
[29] G. Bieber and J. Carpenter. Introduction to service-oriented programming (rev 2.1). 2001.
[30] S. Biswas, M. Zhang, M. D. Bond, and B. Lucia. Valor: efficient, software-only region conflict exceptions. *ACM SIGPLAN Notices*, 50(10):241–259, 2015.
[31] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, pages 107–117, 1998.

[32] D. M. Bulla and V. Udupi. Cloud Billing Model: A Review. In *International Journal of Computer Science and Information Technologies*, pages 1455–1458, 2014.

[33] T. Cerný, M. J. Donahoo, and J. Pechanec. Disambiguation and comparison of soa, microservices and self-contained systems. In *Proceedings of the International Conference on Research in Adaptive and Convergent Systems, RACS'17*, pages 228–235, 2017.

[34] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[35] W. Chuang, B. Sang, S. Yoo, R. Gu, M. Kulkarni, and C. E. Killian. EventWave: Programming Model and Runtime Support for Tightly-coupled Elastic Cloud Applications. In *Proceedings of the 4th ACM Symposium on Cloud Computing, SoCC'13*, pages 21:1–21:16, 2013.

[36] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)*, 31(3):8, 2013.

[37] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles, SOSP'07*, pages 205–220, 2007.

[38] B. Ding, L. Kot, A. J. Demers, and J. Gehrke. Centiman: elastic, high performance optimistic concurrency control by watermarking. In *Proceedings of the Sixth ACM Symposium on Cloud Computing, SoCC'15*, pages 262–275, 2015.

[39] A. Ellis. Introducing Functions as a Service (OpenFaaS). https://blog.alexellis.io/introducing-functions-as-a-service/.

[40] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'19*, pages 3–18, 2019.

[41] Z. Gong, X. Gu, and J. Wilkes. PRESS: predictive elastic resource scaling for cloud systems. In *Proceedings of the 6th International Conference on Network and Service Management, CNSM'10*, pages 9–16, 2010.

[42] M. L. Guimarães and A. R. Silva. Improving early detection of software merge conflicts. In *Proceedings of the 34th International Conference on Software Engineering*, pages 342–352. IEEE Press, 2012.

[43] H. S. Gunawi, T. Do, P. Joshi, P. Alvaro, J. M. Hellerstein, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, K. Sen, and D. Borthakur. Fate and destini: A framework for cloud recovery testing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, 2011.

[44] P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theor. Comput. Sci.*, 410(2-3):202–220, 2009.

[45] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Serverless computation with openlambda. *Elastic*, 60:80, 2016.

[46] B. Jennings and R. Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, 23(3):567–619, Jul 2015.

[47] B. K. Kasi and A. Sarma. Cassandra: Proactive conflict minimization through optimized task scheduling. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 732–741, 2013.

[48] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: a system for dynamic load balancing in large-scale graph processing. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys'16. ACM, 2013.

[49] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. *;login:*, 44(1), 2019.

[50] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 135–146, 2010.

[51] C. McCaffrey. Architecting and launching the halo 4 services. In *USENIX Association*, Santa Clara, CA, 2015.

[52] P. Moritz, R. Nishihara, S. Wang, A. Tumanov, R. Liaw, E. Liang, M. Elibol, Z. Yang, W. Paul, M. I. Jordan, and I. Stoica. Ray: A distributed framework for emerging AI applications. In A. C. Arpaci-Dusseau and G. Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI'18*, pages 561–577, 2018.

[53] A. Newell, G. Kliot, I. Menache, A. Gopalan, S. Akiyama, and M. Silberstein. Optimizing distributed actor systems for dynamic interactive services. pages 38:1–38:15, 2016.

[54] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes. AGILE: elastic distributed resource scaling for infrastructure-as-a-service. In *10th International Conference on Autonomic Computing, ICAC'13*, pages 69–82, 2013.

[55] OASIS. OASIS SOA Reference Model TC. https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=soa-rm.

[56] M. Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[57] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. 2010.

[58] Q. Pu, S. Venkataraman, and I. Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In J. R. Lorch and M. Yu, editors, *16th USENIX Symposium on Networked Systems Design and Implementation, NSDI'19*, pages 193–206, 2019.

[59] J. M. Pujol, V. Erramilli, G. Siganos, X. Yang, N. Laoutaris, P. Chhabra, and P. Rodriguez. The little engine(s) that could: scaling online social networks. In *Proceedings of the ACM SIGCOMM 2010 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 375–386, 2010.

[60] S. Salihoglu and J. Widom. Gps: a graph processing system. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, page 22. ACM, 2013.

[61] B. Sang, G. Petri, M. S. Ardekani, S. Ravi, and P. T. Eugster. Programming scalable cloud services with AEON. In *Proceedings of the 17th International Middleware Conference*, pages 16:1–16:14, 2016.

[62] B. Sang, S. Ravi, G. Petri, M. Najafzadeh, M. S. Ardekani, and P. Eugster. Programmable elasticity for actor-based cloud applications. In *Proceedings of the 9th Workshop on Programming Languages and Operating Systems*, PLOS'17, pages 15–21, 2017.

[63] M. Serafini, E. Mansour, A. Aboulnaga, K. Salem, T. Rafiq, and U. F. Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *PVLDB*, 7(12):1035–1046, 2014.

[64] R. Taft, E. Mansour, M. Serafini, J. Duggan, A. J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing. *PVLDB*, 8(3):245–256, 2014.

[65] S. Wang, I. Keivanloo, and Y. Zou. How do developers react to restful api evolution? In *International Conference on Service-Oriented Computing*, pages 245–259. Springer, 2014.

[66] X. Wu, L. Zhang, Y. Wang, Y. Ren, M. Hack, and S. Jiang. zexpander: A key-value cache with both high performance and fewer misses. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys'16, pages 14:1–14:15, 2016.

[67] M. Zur Muehlen, J. V. Nickerson, and K. D. Swenson. Developing web services choreography standards–the case of REST vs. SOAP. *Decision Support Systems*, 40(1):9–29, 2005.