CAF - The C++ Actor Framework for Scalable and Resource-efficient Applications

Dominik Charousset
Hamburg University of Applied Sciences
dcharousset@acm.org

Raphael Hiesgen
Hamburg University of Applied Sciences
raphael.hiesgen@haw-hamburg.de

Thomas C. Schmidt
Hamburg University of Applied Sciences
t.schmidt@ieee.org

Abstract

The actor model of computation has gained significant popularity over the last decade. Its high level of abstraction combined with its flexibility and efficiency makes it appealing for large applications in concurrent and distributed regimes.

In this paper, we report on our work of designing and building CAF, the "C++ Actor Framework". CAF targets at providing an extremely scalable native environment for building high-performance concurrent applications and distributed systems. Based on our previous library `libcppa`, CAF significantly extends its scopes of application and operation, as well as the range of scalability. The particular contributions of this paper are threefold. First we present the design and implementation of a type-safe messaging interface for actors that rules out a category of runtime errors and facilitates robust software design. Second we introduce a runtime inspection shell as a first building block for convenient debugging of distributed actors. Finally we enhance the scheduling facilities and improve scaling up to high numbers of concurrent processors. Extensive performance evaluations indicate ideal runtime behaviour for up to 64 cores at very low memory footprint. In these tests, CAF clearly outperforms competing actor environments.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent programming; C.2.4 [*Distributed Systems*]: Distributed applications; D.3.4 [*Processors*]: Run-time environments

Keywords Actor Model, C++, Message-oriented Middleware, Distributed Debugging

1. Introduction

In recent times, an increasing number of applications requires very high performance for serving concurrent tasks. Hosted in elastic, virtualized environments, these programs often need to scale up instantaneously to satisfy high demands of many simultaneous users. Such use cases urge program developers to implement tasks concurrently wherever algorithmically feasible, so that running code can fully adapt to the varying resources of a cloud-type setting. However, dealing with concurrency is challenging and handwritten synchronisations easily lack performance, robustness, or both.

At the low end, the emerging Internet of Things (IoT) pushes demand for applications that are widely distributed on a fine granular scale. Such loosely coupled, highly heterogeneous IoT environments require lightweight and robust application code which can quickly adapt to ever changing deployment conditions. Still, the majority of current applications in the IoT is built from low level primitives and lacks flexibility, portability, and reliability. The envisioned industrial-scale applications of the near future urge the need for an appropriate software paradigm that can be efficiently applied to the various deployment areas of the IoT.

Forty years ago, a seminal concept to the problems of concurrency and distribution has been formulated in the actor model by Hewitt, Bishop, and Steiger [15]. With the introduction of a single primitive—called actor—for concurrent and distributed entities, the model separates the design of a software from its deployment at runtime. The high level of abstraction offered by this approach combined with its flexibility and efficiency makes it highly attractive for today's elastic multicore systems, as well as for tasks distributed on Internet scale. As such, the actor concept is capable of providing answers to urgent problems throughout the software industry and has been recognized as an important tool to make efficient use of the infrastructure.

On its long path from an early concept to a wide adoption in the real world, many contributions were needed in both, conceptual modeling and practical realization. In his seminal work, Agha [1] introduced mailboxing for the message processing of actors, and laid out the fundament for an open, external communication [3]. Actor-based languages like Erlang [4] and frameworks such as ActorFoundry—which is based on Kilim [27]—have been released but remained in specific niches, or vendor-specific environments (e.g., Casablanca [24]). Scala includes the actor-based framework Akka [30] as part of its standard distribution, because the actor model has proven attractive to application developers. The application fields of the actor model also include cluster computing as demonstrated by the actor-inspired framework Charm++ [17]. In our previous work on `libcppa` [10], we introduced a full-fledged C++ actor library to the native domain.

In this work, we report on the enhanced "C++ Actor Framework" (CAF)[1]. CAF has evolved from our previous library `libcppa` with significant improvements and additional capabilities. CAF subsumes components for highly scalable core actor programming, GPGPU Computing, and adaptations to a loose coupling for the IoT [16], It has been adopted in several prominent application environments, among them scalable network forensics [31]. In the present paper, we focus on three contributions.

AGERE! '14, USA.

http://dx.doi.org/10.1145/...

[1] http://www.actor-framework.org

1. We enhance robustness of future actor programming by introducing a type-safe message passing interface design.
2. We elaborate first steps towards a distributed actor debugging by an inspection shell for remote actors.
3. We design, implement, and evaluate a scheduling infrastructure for the actor runtime environment that improves scaling up to high numbers of concurrent processors.

The remainder of this paper is organized as follows. Section 2 discusses related work along with our previous contributions and issues we have identified. In section 3, we present our software design for type-safe messaging interfaces between actors. Our prototype infrastructure for runtime inspection is presented in Section 4. Finally, design and implementation choices of our scalable scheduling platform as well as a practical performance evaluation is presented in Section 5 and Section 6 concludes.

2. Background and Related Work

We believe that writing dynamic, concurrent, and distributed applications using a native programming language such as C++ is ill-supported today. Standardized libraries only offer low-level primitives for concurrency such as locks and condition variables. Using such primitives correctly requires a lot of expert knowledge and can cause subtle errors that are hard to find [23] and a naïve memory layout can severely slow down program execution due to *false sharing* [29]. The support for distribution is even less sufficient and developers often fall back to hand-crafted networking components based on socket-layer communication. Transactional memory—either in software [26] or hardware [14]—and atomic operations can help implementing scalable data structures [13] but neither account for distribution nor for communication between software components nor for dynamic software deployment.

The actor model of computation [15] describes computer programs in terms of independent software entities exchanging messages and addresses fault tolerance in a network-transparent way [5]. The actor is the universal primitive for concurrency and distribution. Incoming messages are buffered in FIFO order using a mailbox and are handled sequentially [1]. Implementations such as Erlang allow actors to skip messages for later retrieval, while other implementations require actors to handle messages in the order of arrival.

2.1 Native Actors

The advent of multi-core machines and the proclaimed end of Moore's law [19] make inter-machine concurrency a necessity. At the same time, native programming languages such as C++ are experiencing a renaissance. Since the clock speed no longer increases significantly, computer programs need to make use of existing resources as efficiently as possible.

Implementations of the actor model traditionally focused on virtualized environments such as the JVM [18], while actor-inspired implementations for native programming languages focus on specific niches. For example, Charm++ is directly aimed at software development for supercomputers and a *chare*—the primitive for currency abstraction in Charm++—only offers a subset of the typical actors characteristics. In our previous work on `libcppa`, we presented the design and implementation of a full-fledged native actor system with a strong emphasis of efficiency and runtime performance. We have presented 1) a lock-free mailbox algorithm with an average of $O(1)$ for both enqueue and dequeue operations, 2) an efficient network layer for dynamic distributed systems, 3) a copy-on-write messaging system minimizing copying operations, and 4) an adaptive runtime system able to integrate heterogeneous hardware components via OpenCL.

Based on feedback from both academia and industry as well as from our own experience with `libcppa`, we have identified limitations we need to address. The dynamic typing and runtime pattern matching for messages contradicts the philosophy of C++ developers that rely on the static, strong type checking. This is not only an acceptance issue. Composing large systems out of small software entities is prone to failure if the compiler is unable to validate the (messaging) interfaces between the components. Furthermore, analyzing, tweaking and debugging distributed actors requires comprehensive support from the runtime system as well as a convenient toolkit to aid software developers in this complex task. Lastly, a comprehensive analysis of our previous implementation revealed an inferior scalability of the runtime system when running on massively parallel hardware platforms.

2.2 Verification and Debugging

Parallel execution and the inherent non-determinism of the actor model render static verification of complex, distributed applications using model checking techniques impossible [25]. Although complexity analysis can help programmers to understand and predict performance [2], static models of an application can neither be used to verify their correctness nor to guarantee certain runtime characteristics. Rather than verifying an application by modeling and verifying each state transition statically, applications can be partially verified on a state-by-state basis using either recorded execution traces or in real-time [8].

When implementing distributed applications, developers usually rely on systematic testing and ad hoc debugging. The actor model aids developers in both cases. Since the actor model requires developers to split the application logic into many independent components, those components can be tested individually. For example, the property-based black-box testing tool "Quviq QuickCheck" [6] demonstrates an approach to reveal obvious and subtle bugs in Erlang applications using controllable random test case generation. When facing the complex task of debugging distributed applications, developers can use a recorded message flow of the distributed execution in a postmortem analysis. This approach has been examined by HP and lead to the development of the distributed debugger Causeway [28]. An alternative approach to tackle the complexity of distributed debugging has been made by Dennis Geels, et. al. by using the recorded message flow to replay messages in order to reproduce erroneous behavior [12].

Both verification and debugging rely on extensive support from the runtime system. In this work, we focus on a runtime inspection architecture as first stepping stone to a framework for debugging, verification, and online performance analysis of distributed actor applications.

3. Type-safe Messaging Interfaces

Traditional message passing systems are often implemented in languages performing dynamic type checking or in strongly typed languages but using a dynamic approach with runtime type checks. Such a dynamic approach hides information from the compiler, thus rendering a static analysis of the messaging interfaces impossible. This validation step is crucial for composing large software systems out of small software entities as developers otherwise need to rely on systematic testing of each integration individually. With CAF, we present a software design for strongly typed messaging interfaces that enables the compiler to verify messaging protocols statically at compile time.

3.1 Defining Messaging Interfaces using Patterns

Actors are defined in terms of the messages it receives and sends. The behavior of an actor is hence specified as a set of messages handlers that dispatch extracted data to associated functions. Defining

such handlers is a common and recurring task in actor programming. Pattern matching facilities as known from functional programming languages have proven to be a powerful, convenient and expressive way to define such message handlers. Since C++ does not provide pattern matching facilities, we have decided to implement an internal domain-specific language (DSL) for C++. This DSL is limited to actor messages, because a solution for arbitrary data structures cannot be implemented without a language extension. Unlike other runtime dispatching mechanisms, our pattern matching implementation discloses all types of incoming messages as well as the type of outgoing messages to the compiler. In this way, the compiler can derive the interface of an actor from the definition of its behavior.

Whenever an actors does not want the compiler to derive a messaging interface from its behavior definition, it can store a pattern using instances of the type `behavior`. This type-erasure step is always performed for the dynamic actors we have introduced in our previous work on `libcppa` [10].

A match expression, i.e., the definition of a partial function, usually begins with a call to the function `on` that returns an intermediate object providing the operator ">>". The right-hand side of the operator denotes a callback—usually a lambda expression—which should be invoked after a tuple matches the types given to `on`. The example below illustrates four different ways of defining identical match expressions.

```
on<int>() >> [](int i) { ... },      // (1)
on(val<int>) >> [](int i) { ... },   // (2)
on(arg_match) >> [](int i) { ... }, // (3)
[](int i) { /*...*/ }                // (4)
```

Example (1) illustrates how `on` can be used with template parameters. When using the arguments-only notation, `val<int>` can be used to match any value of type `int`, as shown in (2). To simply infer the types from the signature of the callback, `arg_match` can be used. Usually, `arg_match` is not used as shown in (3), because the constructor of `message_handler` will infer the types from the signature of the callback automatically–as shown in (4)–if it has not been constrained. Rather, `arg_match` can be used as last argument to infer the remaining types from the callback. For example, `on(atom("add"), arg_match) >> [](int, int) {...}` matches only message that consists of an atom of value `"add"` followed by to integers.

It is worth mentioning that our pattern matching implementation behaves as a functional programmer would it expect to, i.e., only the first matched expression is being executed. In our example code above, only (1) could ever been invoked, since it always matches first. Our DSL-based approach has more syntactic noise than a native support within the programming languages itself, for instance when compared to functional programming languages such as Haskell or Erlang. However, we only use ISO C++ facilities, do not rely on brittle macro definitions, and our approach only adds negligible—if any—runtime overhead by making use of expression templates [32].

An important characteristic of our pattern matching engine is its tight coupling with the message passing layer. The runtime system of CAF will create a response message from the value returned from the callback unless it returns `void`. Not only is this convenient for programmers, it also exposes the type of the response message to the type system. This information is crucial to define type-safe messaging interfaces.

It is wort mentioning that we support both function- and class-based actors. The former are implemented as a free function returning the initial behavior for the actor, whereas the first argument denotes the implicit `self` pointer. Class-based actors are derived from either a type-safe or dynamically typed actor base class and must override the virtual member function `make_behavior()` returning the initial behavior. In our examples, we make only use of function-based actors as they require less implementation overhead.

3.2 Strongly Typed Message Interfaces

Dynamically typed actors in CAF use handles of the type `actor`, whereas type-safe actors use handles of type `typed_actor<...>`. The template parameters denote the messaging interface using a series of `replies_to<...>::with<...>` clauses. For example, the following type `testee` identifies an actor that either receives two integers and responds with a single integer or receives a floating pointer number and responds with two floating point numbers.

```
using testee =
  typed_actor<
    replies_to<int, int>::with<int>,
    replies_to<float>::with<float, float>>;
```

When trying to send anything else to an actor of this type, the compiler will reject the code with the error message "typed actor does not support given input".

However, the example above is not an idiomatic typed messaging interface. Since the actor receives primitive types only, the interface lacks semantic information as to what the receiver is supposed to do with those values. The following example models an actor offering a simple service for addition and subtraction of integer values.

```
struct add_request { int a; int b; };
struct sub_request { int a; int b; };
using math =
  typed_actor<
    replies_to<add_request>::with<int>,
    replies_to<sub_request>::with<int>>;
math::behavior_type f(math::pointer self) {
  return {
   [](const add_request& req) {
     return req.a + req.b;
   },
   [](const sub_request& req) {
     return req.a - req.b;
   }
  };
}
// announce custom types (only once)
announce<add_request>(&add_request::a,
                      &add_request::b);
announce<sub_request>(&sub_request::a,
                      &sub_request::b);
// usage example
math ms = typed_spawn(f);
send(ms, add_request{1, 2}); // ok
send(ms, 1, 2); // compiler error
```

This examples makes use of user-defined message types instead of prefixing values with atoms. The type alias `math::behavior_type` is a type that does not perform the type erasure we have previously seen by assigning the patterns to a `behavior`. Instead, input and output types are exposed to the runtime system and—more importantly—to the compiler. User-defined message types—as showcased in the example—must be announced to the type system of CAF to enable serialization and deserialization at runtime.

Whenever a message type changes, existing code will either still works as expected if merely additional fields where added or the compiler will reject the program and points the programmer to each use of that particular message type individually.

3.3 Dynamic Message Interfaces

To illustrate the trade-offs and differences for typed and untyped actors, we provide an implementation of the example in 3.2 using the dynamically typed API as shown below.

```
behavior f(event_based_actor* self) {
  return {
   on(atom("add"), arg_match)
   >> [](int a, int b) {
     return a + b;
   },
   on(atom("sub"), arg_match)
   >> [](int a, int b) {
     return a - b;
   }
  };
}
// usage example
actor ms = spawn(f);
send(ms, atom("add"), 1, 2); // ok
send(ms, 1, 2); // invalid but compiles
```

The definition of user-defined messaging types is no longer required. Instead, an idiomatic way to add semantic information to a message is by prefixing it with atoms.

3.4 Message Passing Interfaces Summary

A dynamic approach has the benefit of being able to provide a single primitive and actors can encode their acquaintances as list over that primitive type. This resembles the original actor modeling that did not specify how–or even if–actors specify the interface for incoming and outgoing messages. Rather, actors are defined in terms of *names* they use, *access rights* to *acquaintances* they grant, and *patterns* they specify to dispatch on the content of incoming data [15].

With strongly typed actors, the compiler statically verifies the protocols between actors. Hence, the compiler is able to rule out a whole category of runtime errors, because protocol violation cannot occur once the program has been compiled. It is worth mentioning that the compiler does not only verify the correct sending of a message but it also can verify the handling of the result when using `sync_send`. For instance, the following example would be rejected by the compiler.

```
math ms = typed_spawn(f);
sync_send(ms, add_request{10, 20}).then(
  [](float result) {
    // compiler error: math actor will
    // send an int as result, not a float
  }
);
```

When using `sync_send`, the sent message will have a unique ID. The sending actor than can use `.then` to install a message handler that is only used for the response message to that particular ID. Please note that `sync_send` does not imply blocking. "Synchronous" messages use the exact same message passing implementation as asynchronous messages and only add a convenient way to uniquely identify a set of request / response messages.

When using a statically typed system, developers are trading convenience for safety. Since software systems grow with their lifetime and are exposed to many refactoring cycles, it is also likely that the interface of an actor is subject to changes. This is equivalent of the schema evolution problem in databases: once a single message type–either input or output–changes, developers need to locate and update all senders and receivers for that message. When introducing a new kind of message to the system, developers also need to identify and update all possible receivers by hand.

With CAF, we lift the type system of C++ and make it applicable to the interfaces of actors. At the same time, we are aware of the fact that dynamically typed systems do have their benefits and that these approaches are not mutually exclusive. Rather, we believe a co-existence between the two empowers developers to make the ideal tradeoff between flexibility and safety. Hence, we have implemented a hybrid system with CAF. Type-safe and dynamic message passing interfaces are equally well supported and interaction between type-safe and dynamic actors is not restricted in any way.

It is up to the architect of a software system to choose when to make use of untyped actors and when to pay the initial programming overhead for typed actors. As a general recommendation we can give based on our experiences with CAF, typed actors should be used for any kind of actor that *can* have non-local dependencies. Such actors are usually central components of a larger system and offer a service to a set of actors that is either not known at coding time or might grow in the future. Type-safe messaging interfaces allow the compiler to keep track of non-local dependencies that exist between central actors and a—possibly large—set of clients. Whenever all possible acquaintances of an actor are known at coding time and if this set of actors is tightly coupled—ideally only exist in the same translation unit—untyped actors are usually a good choice, because they reduce code size.

4. Runtime Inspection

Debugging of distributed systems is inherently complex and well known as a hard problem. In addition to difficulties that derive from concurrent control loops within applications, distribution adds a messaging layer to the list of challenges. Monitoring distributed messaging including its temporal logic is tedious and requires a complete observation infrastructure.

Actors can detect hard errors by monitoring each other and implement recovery strategies, but this mechanism does not provide software developers with sufficient intelligence to understand the cause of an error. The (possibly correlated) state of an incident remains invisible. Further, this mechanism does not help developers in finding inefficiencies or bottlenecks in their software architecture. In general, distributed systems easily attain non-trivial coincident conditions that are harmful, but hard to find without proper tool support.

The first building block required for implementing a high-level, convenient toolkit for debugging is a runtime inspection infrastructure. This infrastructure must provide a full view on crucial information of the distributed system to allow for understanding the runtime behavior of an application. In particular, it must reveal the state of distribution, interconnection and messaging of all participating nodes. In this work, we make the collected information available to developers by complementing the runtime inspection components with an interactive shell.

4.1 Collecting Events in a Distributed System

Figure 1 illustrates the components of our runtime inspection infrastructure. It consists of 1) one configurable Probe at each node that collects events and aggregates statistics of individual processes, 2) a Nexus that receives events from Probe instances and makes them accessible to others, and 3) one or several front-end applications that query the Nexus and can subscribe to events. In this work, we present an interactive shell for a basic inspection front end. The Probes as well as the Nexus are modeled and implemented as actors and communicate via message passing. By monitoring each other, the Probe is able to detect a temporary failure of the Nexus and periodically tries to reconnect. A Probe failure indicates a disconnect from its node, either due to a program or system failure.

4.1.1 Probes

Probes have access to the network layer of CAF. On startup, Probes receive configuration input from command line arguments as well as a configuration file. The minimal configuration needed to initial-

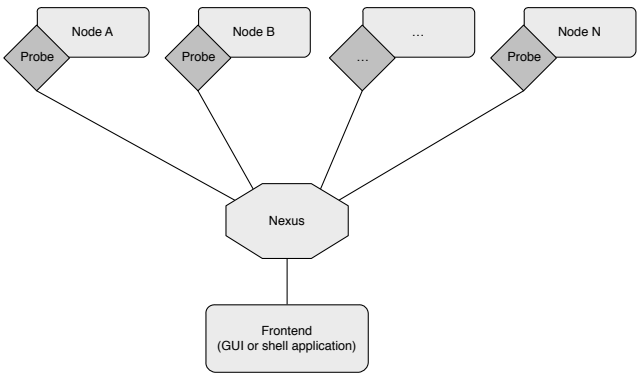


Figure 1. Runtime Inspection Architecture of CAF

ize the probe is the network contact of the Nexus. Probes intercept and forward three kinds of messages to the Nexus.

Activity events are triggered by incoming or outgoing connection and message exchange with actors on different nodes. Forwarding the entire message flow to the Nexus grants maximal transparency, but induces high network traffic and runtime overhead. This information corresponds to a complete logging of the distribution system and can be crucial while investigating erroneous behavior. Whether or not the full message exchange between nodes is protocoled to the Nexus can be configured at runtime.

Error events are triggered by network failures. For instance, when the connection to a node was lost unexpectedly or the delivery of a message has failed because the target node either does not exist or a connection failure occurred during transmission.

Runtime statistics are periodically generated by the Probe and include RAM usage, CPU load, and the number of currently active actors. In this way, observers can spot uneven distribution of work load in the distributed system and can react to over-utilization of the distributed system, e.g., by adding more nodes.

4.1.2 The Nexus

The Nexus provides a global view of the distributed system. It receives and collects events as well as runtime statistics from all Probes and forwards them to front-end applications (clients). The Nexus uses only type-safe messaging interfaces to communicate to its clients and Probes, and statefully manages the configuration of new Probes. New clients subscribe at the Nexus by sending an `add_listener` message and in turn receive all messages in a transparent way.

The Nexus also receives messages to configure the verbosity of the probes, e.g., to enable or restrict the full mirroring of communication between nodes. Configuration messages are broadcasted to all Probes in order to guarantee a consistent view over all nodes. Furthermore, the Nexus serves as network hub for its clients by exposing all of its connections. Hence, clients can send messages to individual nodes or actors that are transparently forwarded by the Nexus. This allows front-end applications to not only observe the system but to interact with it.

4.1.3 Front-end Applications

With our design of the runtime inspection framework, we want to enable front-end applications falling in the following three main categories.

Observing autonomous agents — monitor a distributed systems and verify that it is running within its specified parameters. An example for this kind of application is an automated alert system. Users of such a system may specify thresholds and rules to trigger system alerts. Those rules could query the throughput or load of (parts of) the actor system, possibly revealing that the number of requests cannot be handled in time with the available resources. Other characteristic use cases are in stability and reliability monitoring. An alerting system may control how many node or connectivity failures occur prior to alerting a system administrator.

Supervising autonomous agents — interactively monitor and control certain characteristics of the distributed system. This task includes an active manipulation of the system components and allows for immediate reaction on information gathered, instead of only passively observing it. Agents of this kind can for example enable interactive intervention on errors or perform a distributed load balancing by migrating actors from nodes working to capacity to other nodes.

Performance monitoring & visualization — use runtime statistics to extract a meaningful view of the state of the distributed system. Such tools may grant users convenient access to aggregated information about resource usage on each host as well as the current state of deployment at runtime. Our interactive shell is an example application for this use case and gives developers valuable insights about the runtime characteristics of their system. Such live views can be used to refactor the application for better performance while it is still in development, or to optimize the deployment of a system in production.

4.2 An Interactive Inspection Shell

Our runtime inspection infrastructure is a stepping stone towards a debugging tool for distributed actors. Findings bugs or spotting flaws in the architecture of a system is an interactive, iterative process. Hence, our first focus was on writing an interactive shell.

The interactive shell bundled with CAF allows users to interactively traverse through the actor system. It has a *global mode* as well as a *node mode*. The user can query all participating nodes in the actor system by using the command `list-nodes`. This command is available in both modes and prints the full list of nodes. In the same way UNIX shells allow the user to navigate through the file system, our shell enables the user to navigate through the system using the command `change-node`. We store the last visited nodes and enable users to return to the last node they have visited by using `back`.

In the *node mode*, the user can access any relevant information about the node using `statistics`. This command will display a) the hostname, b) the name of the operating system in use, c) the number of currently running actors, d) the CPU load, e) the amount of available and allocated RAM, f) a full list of network interfaces, g) a list of connected nodes (excluding the Nexus), and h) a list of actors on this node that communicated to other actors in the network.

Since the shell itself is an actor, users are also capable of interacting with the system directly. The command `send` will deserialize a message from its arguments and send it to an actor. For example, `send 5 "Hello Actor"` sends the string "Hello Actor" the actor with ID 5 on the current node. The mailbox of the shell is also exposed the actor and its content can be queried by using `mailbox`, which will print the FIFO numbered list of current mailbox entries. The command `dequeue` accesses the full content of a message and also removes it from the mailbox. Alternatively, users can use `pop-front` to print and remove the oldest element from the mailbox.

The shell enables users to interact with the system in a dynamic and convenient fashion. Using the prototype can reveal bottlenecks in the application that can occur if two or more actors have frequent message exchange but are located on different nodes, causing high network traffic and possibly needless overhead. In using the shell, developers can get valuable feedback during the development process. Still, the shell can currently not provide statistics for individ-

ual actors such as execution time, mailbox contents, idle times, etc. Collecting scheduling-related information in the Probe is part of ongoing and future work.

5. Scheduling Infrastructure

The design of CAF aims at scaling to millions of actors on hundreds of processors. At a first glance, it seems straightforward to implement actors using kernel-level threads. Since an operating system schedules threads in a preemptive manner, actors could not starve other actors. Yet, this naïve approach does not scale up to large-scale actor systems, because threads have significant overhead attached to them. Each thread requires its own stack, signal stack, event mask, and other resources in kernel space. Mapping actors onto threads thus contradicts idiomatic patterns of the actor programming paradigm.

In our first lib [10], we addressed this issue by implementing a userspace scheduler based on a thread pool. Actors were modeled as very lightweight threads that either a) have at least one message in their mailbox and are *ready*, b) have no message and are *blocked*, or c) are currently *executed*. A central management component dispatched actors of state *ready* to a thread. This design was an optimization over the naïve approach of coupling actors with threads, as an actor was only assigned to a thread when ready for execution. Pre-allocated threads from the pool were shared with other actors to minimize system overhead.

A first thorough performance evaluation on a 12-core machine revealed that this scheduling policy reaches its maximum performance on eight cores for classical divide & conquer algorithms. Adding additional concurrency increased the runtime again, since the communication overhead in our central management component outweighed the benefit of additional resources.

A centralized scheduling architecture can efficiently schedule tasks—or actors—based on known deadlines [22] even in multiprocessor environments [11]. However, without a priori knowledge about the tasks, a central architecture cannot dispatch tasks more efficiently than a fully dynamic, decentralized approach. It may induce significant runtime overhead for short-lived tasks, though.

5.1 Work Stealing

Work stealing [9] is an algorithm to schedule multithreaded computation using P worker threads, where P is the sum of all available CPU cores. It has been developed as an alternative to work *sharing* scheduling approaches with centralized dispatching. Work stealing replaces the central job queue by P job queues, one individual queue for each worker. Each worker dequeues work items from its own job queue until it is empty. Once this happens, the worker becomes a *thief*, picking one of the other workers—usually at random—as *victim* and tries to *steal* a work item from its queue. This approach drastically reduces the communication between workers, since they work completely independent from each other as long as there is still work remaining in each queue. In consequence, work stealing induces less communication overhead and outperforms work sharing due to its higher scalability for most application scenarios. Moreover, stealing is a rare event for most work loads and implementations should focus on the performance of the non-stealing case [20].

A widely used variant of work stealing is *fork-join*. Fork-join models the work flow of an application in terms of divide & conquer. A task *forks* by dividing a large computation into smaller ones and then *joins* the results of its child tasks. Because tasks do not share state, they can be executed independently and in parallel using a work stealing algorithm. Fork-join has become particularly popular in the Java community [21] and a framework for fork-join scheduling is part of the standard distribution since Java 1.7.

An inherent characteristic of fork-join application is that each task recursively creates new tasks that become smaller and smaller until they become trivial. Consequently, the oldest elements in the job queue of a worker demand large computations that will likely *fork* into smaller computation. Newer tasks have been created by forking from larger computations and the complexity decreases over time.

To exploit this typical behavior, each worker dequeues work items from its own job queue in LIFO order until there is no work item left. Once it becomes a *thief*, it steals the *oldest* element, i.e., it dequeues in FIFO order. In this way, the stolen work item is likely to have a high complexity and to amortize the communication overhead induced by stealing.

The fork-join work flow correlates to the work flow often seen in actor applications. After receiving a task via a message, an actor can divide it into smaller tasks and spawn one new actor per newly created sub task. Because this is a common pattern, newer implementations of the actor model—such as Akka—use this scheduling algorithm per default.

5.2 A Configurable, Policy-based Scheduler Infrastructure

Despite suiting many work loads, work stealing schedulers have limitations. When facing hard real-time requirements, for example, central dispatching based on deadlines is crucial. Furthermore, a priori knowledge about the runtime behavior of certain actors cannot be exploited efficiently in a decentralized system with rigorously restricted communication. Hence, an implementation of the actor model should provide a default scheduling algorithm that fits most application scenarios while allowing users to deploy a custom implementation.

Independently from the scheduling algorithm in use, developers need to balance throughput, fairness, and latency. These three criteria have different impact depending on the application domain. When optimizing for throughput, developers strive to maximize the number of messages a system can handle per second. A fair scheduling, on the other hand, tries to split CPU time evenly among all actors. Lastly, when minimizing latency, developers want to have a short period of time between receiving and handling incoming—usually external—messages. A fair scheduling usually causes low latency although the overhead attached to evening out CPU time can increase latency if system resources are not used efficiently.

These three criteria can be balanced by configuring the number of messages an actor is allowed to handle before returning control back to the scheduler. This can be supplemented by adding a time an actor should *at least* run before returning control to the scheduler. In this way, large amounts of messages that cause only minimal work will not pile up in the mailbox of an actor.

Allowing actors to fully drain their mailbox usually maximizes throughput, because it minimizes scheduling overhead. This approach can nevertheless deliver suboptimal throughput if the actors are running on an intermediate node in a distributed system that consumes several work items via the network and produces new work items that are consumed on different nodes. In such cases, this scheduling strategy can lead to bursts, as arriving work items for currently waiting actors pile up. Although the CPU on one host will have efficient use, other CPUs of subsequent hosts might idle.

On the other extreme, actors would be only allowed to consume one single message at a time before returning control to the scheduler. Combined with a round-robin scheduling, this strategy guarantees a very fair scheduling, given that no actor actively starves others. Still, striving to achieve maximum fairness is not an efficient scheduling strategy in most cases. CAF implements event-based actors and chains message handler invocations rather than performing context switching. Nonetheless, chaining unrelated message han-

dlers causes frequent cache misses by changing the working set constantly and maximizes access to the job queue of each worker.

As a general-purpose framework for actor programming, CAF seeks to cover most use cases with an efficient default implementation. However, this default implementation is exposed to developers to grant them full access to performance-critical components. This includes configuring default implementations as well as replacing them if they do not match the use case of the application.

Developers can install a user-defined scheduler with the function `set_scheduler`.

```
template <class Policy = work_stealing>
void set_scheduler(size_t num_workers = ...,
                   size_t max_msgs = 0);
```

The `num_workers` argument defines how many threads should be allocated. Per default, this value is set to the number of CPU cores found at runtime. The second argument, `max_msgs`, specifies how many messages an actor is allowed to consume before returning control back to the scheduler, whereas 0 means indefinite. The template parameter `Policy` needs to implement the following concept class.

```
struct scheduler_policy {
  struct coordinator_data;
  struct worker_data;
  void central_enqueue(Coordinator*,
                       resumable*);
  void external_enqueue(Worker*,
                        resumable*);
  void internal_enqueue(Worker*,
                        resumable*);
  void resume_job_later(Worker*,
                        resumable*);
  resumable* dequeue(Worker*);
};
```

The scheduler itself consists of a central coordinator and workers. Data fields needed for scheduling are configured using `coordinator_data` and `worker_data`, respectively. Enqueue operations to the coordinator via `central_enqueue` are caused by "top-level" spawns, i.e., actors that have been spawned either from a non-actor context or from a detached actor. Whenever a cooperatively scheduled actor spawns actors, it uses `internal_enqueue` on the worker it is being executed by. Since it is only being called from the thread managed by the worker this enqueue operation does not need to be synchronized. The function `external_enqueue` can be used by the coordinator to delegate an enqueue operation to one of its workers. Actors that have exceeded the number of allowed dequeue operations call `resume_job_later`. Workers use the function `dequeue` to get the next actor in line.

In our default implementation, i.e., `work_stealing`, the coordinator does not have a queue and simply forwards enqueue operations to its workers in round-robin order.

An implementation based on a thread pool could do the opposite, i.e., use a central queue in the coordinator and no data fields in the workers. The `max_msg` parameter allows developers to fine-tune the behavior of our default implementation. Furthermore, the policy-based design enables users to deploy their own scheduling algorithm in case their application domain requires a specialized algorithm tailored to the needs of that particular work load.

5.3 Performance Evaluation

For a scalability study of our scheduling infrastructure, we use the benchmark programs introduced in our previous work [10]. Instead of a 12-core machine, we now employ a host consisting of four 16-core AMD Opteron processors with 2299 MHz each. We vary the number of active CPU cores from 4 to 64 to generate an evenly distributed workload on each processor.

For comparative references, we use the implementations of Erlang, Charm++, ActorFoundry, and Scala with the Akka library. In detail, our benchmarks are based on the following implementations of the actor model: (1) C++ with CAF 0.10 (*CAF*) and Charm++ 6.5.1 (*Charm*), (2) Java with ActorFoundry 1.0 (*ActorFoundry*), (3) Scala 2.10.3 with the Akka library (*Scala*), and (4) Erlang in version 5.10.2 (*Erlang*). CAF and Charm++ have been compiled as release versions using the GNU C++ compiler in version 4.8.1. Scala and ActorFoundry run on a JVM configured with a maximum of 10 GB of RAM using the JDK in version 8. For compiling ActorFoundry, we have used the Java compiler in version 1.6.0_38, since this version is required by its bytecode post-processor.

We measure both clock time and memory consumption. Measurements were averaged over 10 independent runs to eliminate statistical fluctuations. Our results on memory are visualized by box plots to represent its variability in a transparent way. The source code of all benchmark programs are published online at `https://github.com/actor-framework/benchmarks`.

5.3.1 Overhead of Actor Creation

Our first benchmark considers a simple divide & conquer algorithm. It computes 2^{20} by recursively creating actors. In each step N, an actor spawns two additional actors with $N-1$ and waits for the (sub) results of the recursive descend. This benchmark creates more than one million actors, primarily revealing the creation overhead. It is wort mentioning that this algorithm does not imply that a total of one million actors exist at the same time.

Figure 2(a) shows the time to create about a million actors as a function of available CPU cores. Charm++ and CAF scale down nicely with cores, CAF being the fastest implementation with less than a second on twenty or more cores. Erlang can run the benchmark in about three seconds, while Scala fluctuates around 14 seconds and ActorFoundry takes around 19 seconds. For the latter three, CPU gains cease with more than about 12 cores.

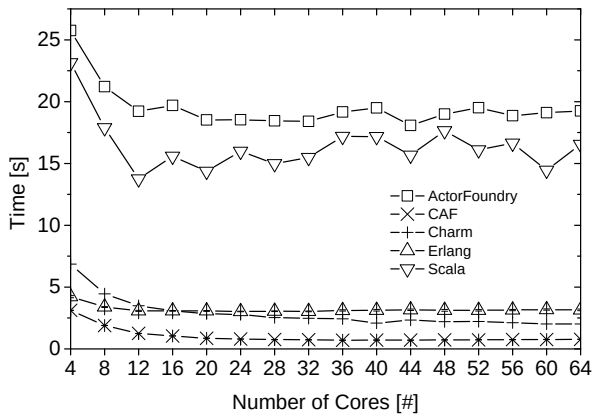
Figure 2(b) shows the memory consumption during the benchmark. Results largely vary in in values and variation. ActorFoundry allocates significantly more memory than all other implementations, peaking around 3.5 GB of RAM with an average of ≈ 1.8GB. Erlang follows with a spike above 2 GB of RAM and an average of ≈ 1 GB. Scala has an average RAM consumption of 500 MB, with a spike about 750 MB. Charm++ and CAF show low values and variations, CAF remaining below 57 MB. This low limit does not imply that an actor uses less than 100 Bytes in CAF. Merely, CAF releases system resources as soon as possible and efficiently re-uses memory from completed actors.

5.3.2 Mailbox Performance in N:1 Communication Scenario

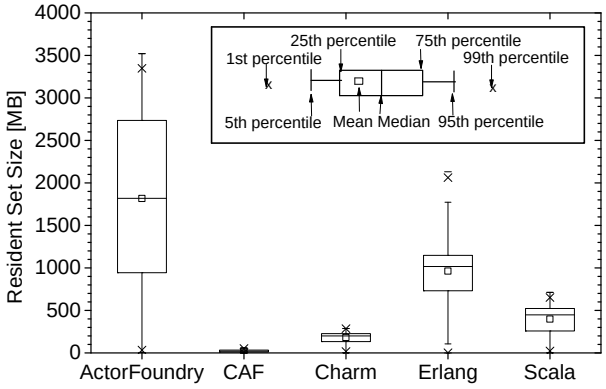
Our second benchmark measures the performance in an N:1 communication scenario. This communication pattern can be observed frequently in actor programs, e.g., whenever an actor distributes tasks by spawning a series of workers and awaits the results.

We use 100 actors, each sending 1,000,000 messages to a single receiver. The minimal runtime of this benchmark is the time the receiving actor needs to process the 100,000,000 messages. It is to be expected that the runtime increases because adding more hardware concurrency increases the probability of write conflicts.

Figure 3(a) visualizes the time consumed by the applications to send and process the 100,000,000 messages as a function of available CPU cores. The processing overhead increases steadily for Erlang from 20 cores onward after reaching a local maximum on 12 cores. Scala and ActorFoundry exhibit a nearly stable behavior up to 36 cores with continuous, fluctuating increase in runtime after that point. Charm++ slightly grows for lower core numbers, and also shows fluctuations for higher concurrency, whereas CAF has outliers on 24 and 32 cores but on the overall remains stable. On

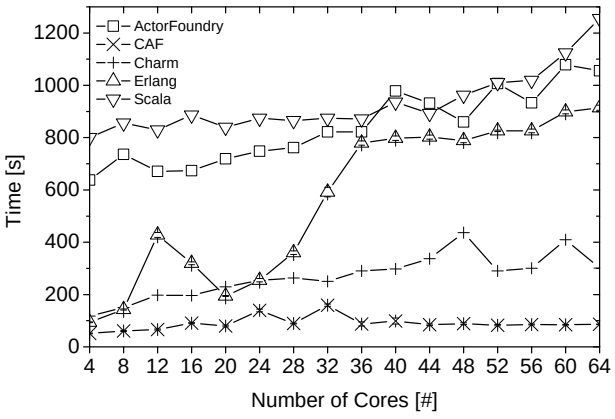


(a) Actor creation time

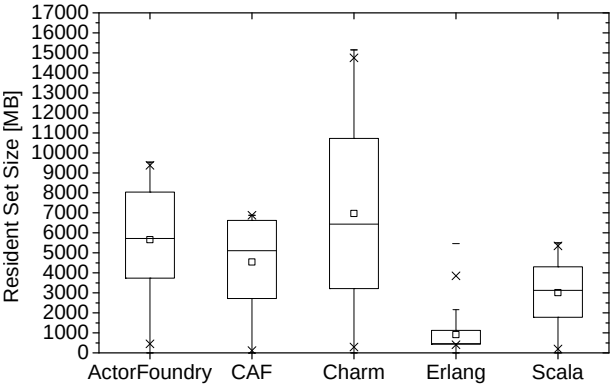


(b) Memory consumption

Figure 2. Actor creation performance for 2^{20} actors

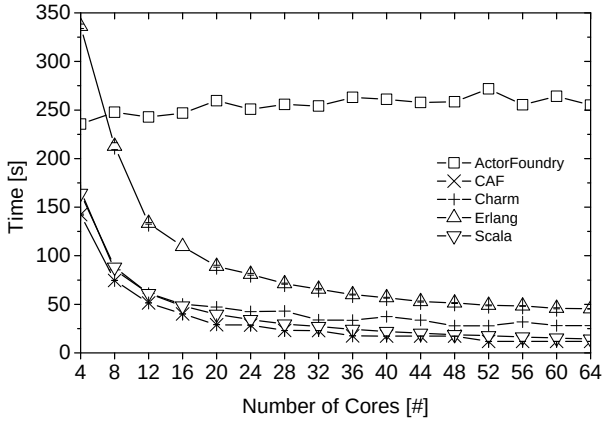


(a) Sending and processing time

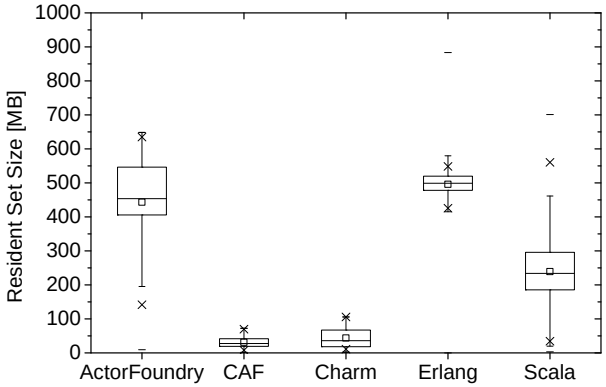


(b) Memory consumption

Figure 3. Mailbox performance in N:1 communication scenario



(a) Sending and processing time



(b) Memory consumption

Figure 4. Performance in a mixed scenario with additional work load

64 cores, CAF has an average runtime of 78 seconds, which is less than a tenth of the 831 seconds of Erlang, or of the 1046 seconds measured for Scala.

Figure 3(b) shows the resident set size during the benchmark execution. In this scenario, a low memory usage can hint to a performance bottleneck, as 100 writers should be able to fill a mailbox faster than one single reader is able to drain it. To minimize memory consumption, an implementation would need to actively slow down senders, which cooperative scheduled systems cannot do without blocking sending threads—a strategy that has the risk of running into a deadlock. The preemptive scheduling strategy of Erlang seems to deliver a good tradeoff between runtime and memory consumption at first, but fails to maintain a reasonable runtime for high levels of hardware concurrency. Scala and ActorFoundry have a high memory consumption while running more than six times slower than CAF on average, indicating that writers do block readers and messages accumulate in the mailbox while the receiver is unable to dequeue them due to synchronization issues.

5.3.3 Mixed Operations Under Work Load

In this benchmark, we consider a realistic use case including a mixture of operations under heavy work load. The benchmark program creates a simple multi-ring topology with a fixed number of actors per ring. A token with an initial value of 1,000 is passed along the ring and decremented in each round. A client that receives the token forwards it to its neighbor and terminates whenever the value of the token is 0. Each of the 100 rings consists of 100 actors and is re-created 4 times. Thus, we continuously create and terminate actors with a constant stream of messages. In addition, one worker per ring performs prime factorization to add numeric work load to the system.

Figure 4(a) shows the runtime behavior as a function of available CPU cores. An ideal characteristic would halve the runtime when doubling the number of cores. All implementations except for ActorFoundry exhibit an almost linear speed-up. The latter remains at a runtime around 250 seconds. Since it never uses more than 500% CPU at runtime, a better scalability cannot be expected. For the first time, Akka is faster than Erlang and Charm++, though it still does not reach the performance of CAF. The performance gap between Erlang and Scala results from the fact that the JVM performs the prime factorization at the same speed as C++ while the VM of Erlang is about 50% slower.

Figure 4(b) shows the memory consumption during the mixed scenario. Qualitatively, these values coincide well with our first benchmark results on actor creation. CAF again has a very constant and thus predictable memory footprint, while using significantly fewer memory than all other implementations, i.e., less than 72 MB.

5.3.4 Discussion

ActorFoundry is the only implementation under test that is unable to utilize hardware concurrency efficiently in our mixed case benchmark. In all three tested scenarios, CAF runs faster than Charm++ and uses less memory despite both being implemented in C++. It is worth mentioning though that Charm++ focuses on performance on clusters and supercomputers and as a direct result might not be as optimized for single-host performance. Still, there are overlapping use cases for those systems that make a comparison justifiable. For our runtime comparison, we have used the standalone version of Charm++ instead of its `charmrun` launcher that can be used to distribute an application or parallelize it using processes.

On the overall, CAF is faster than any other implementation in our three use cases and always scales almost ideally up to 64 cores. CAF uses significantly less memory than virtualized implementations, but also outperforms the native Charm++ system. The mailbox performance benchmark is the only case where CAF consumes more memory than Erlang and Scala. However, the high memory allocation is a direct result of its highly scalable, lockfree mailbox implementation and allows CAF to outperform competing implementation by orders of magnitude on 64 cores.

6. Conclusions and Outlook

Currently the community faces the need for software environments that provide high scalability, robustness, and adaptivity to concurrent as well as widely distributed regimes. In various use cases, the actor model has been recognized as an excellent paradigmatic fundament for such systems, but there is still a lack of full-fledged programming frameworks. The latter holds in particular for the native domain.

In this paper, we presented CAF, the "C++ Actor Framework". CAF scales up to millions of actors on many dozens of processors including GPGPUs, and down to small systems (like Rasberry PIs) in loosely coupled environments as are characteristic for the IoT. We introduced an advanced scheduling core and presented benchmark results of CAF that clearly confirmed its excellent performance. We further reported on our ongoing efforts to make this framework a production tool set for reliable software development: a strongly typed message interface design to reduce error-proneness, and a distributed runtime inspection for monitoring and debugging.

There are four future research directions. Currently, we are reducing the resource footprint of CAF even further and port to the micro-kernel IoT operating system RIOT [7]. Second we work on extending scheduling and load sharing to distributed deployment cases and massively parallel systems. Third we will extend our design to achieve more effective monitoring and debugging facilities. Finally, a robust security layer is on our schedule that subsumes strong authentication of actors in combination with opportunistic encryption.

7. Acknowledgements

The authors would like to thank Alex Mantel and Marian Triebe for their helping hands in implementing benchmark and runtime inspection programs as well as testing and bugfixing. We further would like to thank the iNET working group for vivid discussions and inspiring suggestions. Funding by the German Federal Ministry of Education and Research within the projects SAFEST and ScaleCast is gratefully acknowledged.

References

[1] G. Agha. Actors: A Model Of Concurrent Computation In Distributed Systems. Technical Report 844, MIT, Cambridge, MA, USA, 1986.

[2] G. Agha and W. Kim. Parallel programming and complexity analysis using actors. In *Massively Parallel Programming Models, 1997. Proceedings. Third Working Conference on*, pages 68–79, Nov 1997.

[3] G. Agha, I. A. Mason, S. Smith, and C. Talcott. Towards a Theory of Actor Computation. In *Proceedings of CONCUR*, volume 630 of *LNCS*, pages 565–579, Heidelberg, 1992. Springer-Verlag.

[4] J. Armstrong. Erlang - A Survey of the Language and its Industrial Applications. In *Proceedings of the symposium on industrial applications of Prolog (INAP96)*, pages 16–18. Hino, October 1996.

[5] J. Armstrong. *Making Reliable Distributed Systems in the Presence of Software Errors*. PhD thesis, Department of Microelectronics and Information Technology, KTH, Sweden, 2003.

[6] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang*, ERLANG '06, pages 2–10, New York, NY, USA, 2006. ACM. ISBN 1-59593-490-1.

[7] E. Baccelli, O. Hahm, M. Günes, M. Wählisch, and T. C. Schmidt. RIOT OS: Towards an OS for the Internet of Things. In *Proc. of the 32nd IEEE INFOCOM. Poster*, Piscataway, NJ, USA, 2013. IEEE Press.

[8] H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based Runtime Verification. In *Verification, Model Checking, and Abstract Interpretation*, pages 44–57. Springer, 2004.

[9] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, Sept. 1999.

[10] D. Charousset, T. C. Schmidt, R. Hiesgen, and M. Wählisch. Native Actors – A Scalable Software Platform for Distributed, Heterogeneous Environments. In *Proc. of the 4rd ACM SIGPLAN Conference on Systems, Programming, and Applications (SPLASH '13), Workshop AGERE!*, New York, NY, USA, Oct. 2013. ACM.

[11] M. Dertouzos and A. Mok. Multiprocessor Online Scheduling of Hard-Real-Time Tasks. *Software Engineering, IEEE Transactions on*, 15(12):1497–1506, Dec 1989. ISSN 0098-5589.

[12] D. M. Geels, G. Altekar, S. Shenker, and I. Stoica. *Replay Debugging for Distributed Applications*, volume 68. 2006.

[13] M. Herlihy. Wait-Free Synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[14] M. Herlihy and J. E. B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th ISCA*, pages 289–300, New York, NY, USA, May 1993. ACM.

[15] C. Hewitt, P. Bishop, and R. Steiger. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd IJCAI*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[16] R. Hiesgen, D. Charousset, and T. C. Schmidt. Embedded Actors – Towards Distributed Programming in the IoT. In *Proc. of the 4th IEEE Int. Conf. on Consumer Electronics - Berlin*, ICCE-Berlin'14, Piscataway, NJ, USA, Sep. 2014. IEEE Press.

[17] L. V. Kale and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. Technical report, Champaign, IL, USA, 1993.

[18] R. K. Karmani, A. Shali, and G. Agha. Actor Frameworks for the JVM Platform: A Comparative Analysis. In *PPPJ*, pages 11–20, 2009.

[19] L. B. Kish. End of Moore's law: Thermal (Noise) Death of Integration in Micro and Nano Electronics. *Physics Letters A*, 305(3–4):144–149, 2002. ISSN 0375-9601.

[20] V. Kumar, D. Frampton, S. M. Blackburn, D. Grove, and O. Tardieu. Work-stealing Without the Baggage. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 297–314, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6.

[21] D. Lea. A Java Fork/Join Framework. In *Proceedings of the ACM 2000 Conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM. ISBN 1-58113-288-3.

[22] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20(1):46–61, Jan. 1973. ISSN 0004-5411.

[23] S. Meyers and A. Alexandrescu. C++ and the Perils of Double-Checked Locking. *Dr. Dobb's Journal*, July 2004.

[24] Microsoft. Casablanca. `http://casablanca.codeplex.com/`, 2012.

[25] K. Sen, A. Vardhan, G. Agha, and G. Rosu. Efficient Decentralized Monitoring of Safety in Distributed Systems. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 418–427, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2163-0.

[26] N. Shavit and D. Touitou. Software Transactional Memory. In *Proceedings of the fourteenth annual ACM symposium on PODC*, pages 204–213, New York, NY, USA, 1995. ACM.

[27] S. Srinivasan and A. Mycroft. Kilim: Isolation-Typed Actors for Java. In *Proceedings of the 22nd ECOOP*, volume 5142 of *LNCS*, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.

[28] T. Stanley, T. Close, and M. S. Miller. Causeway: A message-oriented distributed debugger. Technical report, Technical Report HPL-2009-78, HP Laboratories, 2009.

[29] J. Torrellas, H. S. Lam, and J. L. Hennessy. False Sharing and Spatial Locality in Multiprocessor Caches. *IEEE Trans. Comput.*, 43(6):651–663, June 1994.

[30] Typesafe Inc. Akka. `akka.io`, March 2012.

[31] M. Vallentin, D. Charousset, T. C. Schmidt, V. Paxson, and M. Wählisch. Native Actors: How to Scale Network Forensics. In *Proc. of ACM SIGCOMM, Demo Session*, New York, August 2014. ACM. URL `http://dx.doi.org/10.1145/2619239.26314714`.

[32] T. Veldhuizen. Expression Templates. *C++ Report*, 7:26–31, 1995.