Multitier Reactive Programming in High Performance Computing

Short Paper

Daniel Sokolowski
Technische Universität Darmstadt
Germany
sokolowski@cs.tu-darmstadt.de

Philipp Martens
Technische Universität Darmstadt
Germany
philipp.martens@stud.tu-darmstadt.de

Guido Salvaneschi
Technische Universität Darmstadt
Germany
salvaneschi@cs.tu-darmstadt.de

Abstract

High Performance Computing (HPC) is crucial in a number of sectors, including weather forecasts, particle simulations and fluid dynamics. Existing programming frameworks for HPC expose developers to low-level details such as message passing and explicit memory management, which are hard to program and error-prone.

In this paper, we present ongoing work on increasing the level of abstraction for HPC. We tackle this problem with an approach based on a combination of multitier programming and reactive programming which enables the development of complex processor configurations in a uniform way using event streams as communication pattern. We report our experience with LULESH, a well known HPC benchmark, and we outline our research roadmap.

CCS Concepts • **Software and its engineering** → *Grid computing*; • **Computer systems organization** → *n-tier architectures*;

Keywords High Performance Computing, Reactive Programming, Tierless Programming

1 Introduction

High Performance Computing (HPC) [14] enables running scientific and technical computations at high speed and it is crucial in a number of sectors including weather forecasts, fluidynamics and particle simulation. The programming models adopted in HPC aim to take advantage of modern hardware architectures, which can scale to up to millions of processors in the case of supercomputers. Such programming models are based on languages like C, C++ or Fortran. These feature explicit memory management and get augmented with libraries to execute instructions in parallel on multiple processors as well as to support inter-process communication via message passing (MPI [13]/OpenMP).

While the current approach to program HPC applications maximizes efficiency, it also forces developers to deal with a kind of programming experience deeply tangled with low-level details that in many other areas of software development have been abstracted away by high-level languages.

In this paper, we present our ongoing research on increasing the level of abstraction for programming frameworks in HPC. Our approach stems from the observation that PGAS languages [10] share a number of similarities with multitier languages [6, 7, 27, 32, 33]. The latter, have been recently explored in combination with managed runtimes (e.g., Java and Scala) and reactive abstractions, which promise to conveniently relieve developers from both manual memory management and message passing [35, 36]. We present our preliminary results based on the LULESH HPC benchmark, and discuss a research roadmap to improve the abstraction level of current HPC programming frameworks. In summary, we make the following contributions:

- We analyze multitier programming and reactive programming from the perspective of HPC applications and we show how their combination has the potential to improve the design of HPC software.
- We perform a case study and showcase this approach based on the LULESH benckmark, a real world HPC application consisting of more than 6k lines of C++.
- We identify open problems and we present an outlook for future research on combining multitier programming and reactive programming for HPC.

The paper is structured as follows. In section 2, we motivate and introduce our approach to apply ScalaLoci to HPC, present the framework and point out challenges. Section 3 presents a first case study, which highlights necessary future work. Section 4 shows our preliminary performance results. Section 5 presents our research plans, section 6 discusses related work. Section 7 concludes.

2 Multitier Reactives in HPC

In this section, we explain the potential benefits of adopting multitier reactive programming in HPC, present the frameworks we adopt in this paper, and we analyze the challenges of using these paradigms in the context of HPC.

2.1 A Case for Multiter Reactive Programming

Today's HPC applications require the development of distributed software in which components communicate to exchange partial results of a computation. Explicit handling of code assigned to each processor and of inter-process communication is error-prone and can severely increase the complexity of applications.

Multitier programming languages enable the joint development of distributed system components within one compilation unit. This approach makes the distributed communication flow more explicit and does not break the distributed logic of the application at the network boundaries. In addition, multitier programming reduces the code overhead for communication, because communication code is injected by the compiler. This solution allows developers to focus on the application logic rather than low-level details like data transmission among hosts.

Ideally, the execution of components in distributed systems is parallelizable. In this context, flexibility in expected execution times is crucial to avoid waiting time. Asynchronous communication enables this kind of flexibility. Reactive programming provides support for asynchronous communication and it improves code readability regarding execution and data flows [22].

These observations suggest to combine multitier languages and reactive programming for HPC applications. As suitable multitier language we identified ScalaLoci, because it supports multitier programming in a high-level language (Scala) in user-defined, complex system architectures. Moreover, through the integration of REScala, ScalaLoci features multitier reactive programming across nodes. These two libraries are briefly introduced in the following.

2.2 ScalaLoci

ScalaLoci [35, 36] is a general purpose multitier language, which supports various architectures of distributed systems. Those are specified by *peer types*, which represent endpoints, e.g., processes, of the application. Their connections are specified by so called *ties*, which setup bindings and their arities as *optional*, *single* or *multiple*. These are examples for a homogeneous peer-to-peer and a client-server architecture:

```
// peer-to-peer
@peer type Node <: { type Tie <: Multiple[Node] }
// client-server
@peer type Server <: { type Tie <: Multiple[Client] }
@peer type Client <: { type Tie <: Single[Server] }
```

Based on these peers, ScalaLoci allows the located placement of values and functions using *placement types*. These augment a data type with the peer type, on which the value or function is placed on. At compile time, the code is split up for the various executables of the program along these types. Moreover, ScalaLoci checks statically, whether remote function calls and value transports are correct according to the architecture specification. This makes the dataflow visible and enforces correctness across processes. Examples for the placement, remote value accesses and remote procedure calls are shown in Listing 1.

```
// central public information on Server
val dailyMsg: String on Server = placed { "It's Friday" }
// central local information on Server
val secret: String localOn Server = placed { "XD" }
// access central information on Client
val centralMsg: Future[String] on Client =
placed { dailyMsg.asLocal }
// access local information, shows error in IDE and does not compile
val centralSecret: Future[String] on Client =
placed { secret.asLocal } // privacy violation
// print hello on a client
def sayHi(): Unit on Client = placed{ println("Hello") }
// print hello on all clients from server
placed[Server] { remote call sayHi() }
```

Listing 1. ScalaLoci placement types, remote value accesses and remote procedure calls on a client-server architecture.

While ScalaLoci enables the integration of distributed control flows, recent versions also introduced a module concept. The modularization can be used to separate sub-systems in the program from each other. The crucial advantage over other module systems is that these separation boundaries do not have to be at the inter-process communication points, but can be chosen in a problem oriented and domain specific way to encapsulate a distributed functionality into a single multitier module.

We see great potential in using ScalaLoci for HPC applications, because system architecture and modularization can be performed in a domain specific way, which is much more intuitive than today's HPC programming techniques. Moreover, the control and data flow are visible across peers. This is further improved through reactive abstractions of which we describe the applied implementation next.

2.3 REScala

REScala [29] is a Scala library, which integrates reactive values seamlessly with an event system. REScala provides data types for *events* (Listing 2) and so called *signals* (Listing 3). Events can be used to propagate changes imperatively. Signals update themselves and propagate changes automatically whenever one of their functional dependencies changes.

Both abstractions have the advantage of helping the developer to write encapsulated and composable code, with a clear visualization of the data flow [30, 31]. Moreover, the implied update propagation of values can safe a lot of manual implementation work for the exchange of changing data.

REScala events and signals are available for ScalaLoci and can be used seamlessly across peers. This enables simple

```
val fruits = Evt[String]()          // Event declaration
fruits.observe(println(_))
fruits.fire("Apple")                // Prints "Apple"
fruits.fire("Banana")               // Prints "Banana"
```

Listing 2. REScala event example.

```
val name = Var("Anna")              // Signal value declaration
val s = Signal {"Hey " + name()}    // Dependent signal declaration
s.changed.observe(println(_))       // Prints "Hey Anna" directly
name.set("Berta")                   // Prints "Hey Berta"
name.set("Chris")                   // Prints "Hey Chris"
```

Listing 3. REScala signal example.

implementation of complex data flows in HPC programs. Thereby, the communication is asynchronous and potentially allows the implementation of algorithms that do not waste computational time due to barrier synchronization.

2.4 Challenges of Multitier Reactives

These are the challenges we see for multitier reactive programming in HPC.

Architecture Representation. In traditional programming languages the code that belongs to different peers is developed separately, forcing the separation of logic at inter-process communication and introducing error-prone boilerplate code for communication. MPI aims to simplify inter-process communication with a message passing model that is especially helpful in *Single Program Multiple Data* (SPMD) applications. However, this solution addresses processes via (multidimensional) indexes, which is very abstract and not related to the underlying architecture of the program. This complexity can cause undetected and hard-to-debug errors in the code. In contrast, multitier abstractions allow domain specific architecture definitions, which improve the understandability of the communication code and introduce static correctness guarantees for inter-process communication.

Asynchronous Control Flow. MPI assumes a coordinated startup of all processes and is not designed to scale the system dynamically or support fault-tolerance mechanisms. Traditionally, most HPC applications do not require those features, but recent and future hardware developments lead to HPC application deployments onto hardware clusters, in which the probability of a partial hardware failure is higher. Also, the execution speed variability of today's hardware processors is significant. In parallel applications written in a synchronous style – like many MPI and OpenMP applications are – this leads to wasted computational time at faster CPUs at each synchronization barrier. Reactive programming can greatly help addressing these issues, providing abstractions for work decomposition into asynchronous tasks with asynchronous communication patterns that can account for varying execution times on distributed processors and avoid unnecessary synchronization barriers.

Performance. A disadvantage of a high-level language with virtualization and garbage collection is that it is usually less performant than hand-optimized low-level C code. For a first step, we consider to implement often executed core functions in optimized C and orchestrate them using ScalaLoci by leveraging the *Java Native Interface* (JNI). We believe this might be a valid and practicable compromise to combine the best of both approaches. In further steps we can also consider heap management optimizations or manual memory management in the JVM as done for HPC, e.g., in [3].

3 ScalaLoci in HPC

In this section we introduce the LULESH Benchmark and discuss our implementation of LULESH using the ScalaLoci multitier programming language.

3.1 The LULESH Benchmark

Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) [18] is a hydrodynamics proxy application [17] developed at *Lawrence Livermore National Laboratory* (LLNL). LULESH is a smaller, but representative example of scientific applications in HPC, including the way it implements numerical algorithms and data transfer among processors. Initially developed as a problem for the DARPA UHPC program, which aimed for sustaining the design of extreme scale computers, it is also commonly used as case study in HPC software design, e.g., by [15, 16, 21].

LULESH implementations exist for many technologies. In our work, we refer to the baseline implementation for CPU based computation in the version 2.0.3. This code is implemented in C++ and supports single-threaded sequential execution, multi-threaded execution using OpenMP, distributed execution with MPI, and the combination of these modes. Thus, it is a suitable reference for various setup sizes.

After initialization, the simulation in LULESH is achieved by a central main loop. Each iteration simulates a discrete time step and consists of decentralized calculations for the state update of the simulated nodes and elements as well as the size of the simulated time step. The execution terminates after a predefined timespan has been simulated.

Topology and Communication. LULESH simulates expansion in 3-dimensional space. Thereby, the entire simulation space is a cube. For shared-nothing parallelization this cube

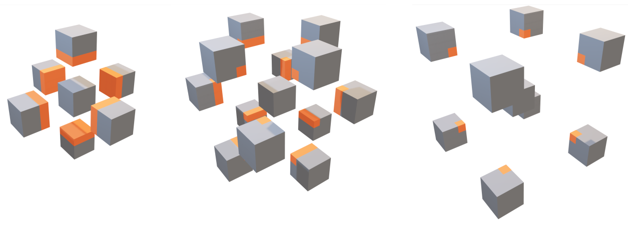

Figure 1. Grid architecture in the LULESH benchmark.

is sub-divided into a mesh of hexahedrons, which are initially equally sized cubes, but deform over time. All processes simulate one hexahedron, called a *domain*. The number of processes n has to be $l^3, l \in \mathbb{N}_{>0}$.

In each iteration, all processes need to decide on a common time step, which requires global communication among all processes. Furthermore, processes exchange node or element data with the processes simulating adjacent domains. Through the 3-dimensional arrangement into *planes*, *rows* and *columns*, a domain exchanges data with up to 26 other domains over different *communication directions*: 6 faces, 12 edges and 8 corners (Figure 1).

3.2 ScalaLoci Implementation

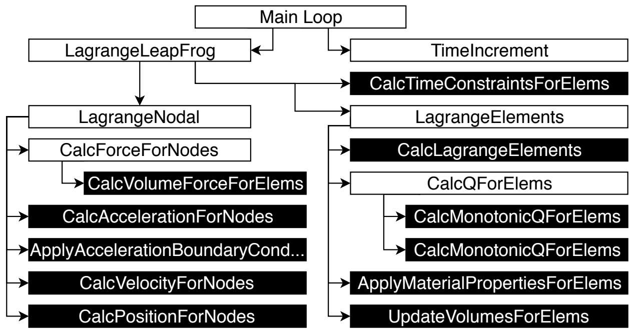

Figure 2. High-level call graph of LULESH.

We ported the main program logic, which contains all inter-process communication, to ScalaLoci while keeping the optimized computation kernels in C++. The detailed separation is visualized in Figure 2, which resembles a high-level call graph of the baseline LULESH implementation. The white methods are re-implemented in ScalaLoci, while the black methods are kept in C++ and called via JNI. Thereby, OpenMP based parallelization is kept in the C++ methods.

Communication Setup. A core advantage of the ScalaLoci adaption is the explicit definition of the peer architecture. For the global communication, we define a single `Master` peer, while all processes simulating a domain are abstract `Worker` peers connected to `Master`:

```
@peer type Master <: { type Tie <: Multiple[Worker] }
@peer type Worker <: { type Tie <: Single[Master] }
```

Moreover, for each of the possible 26 communication directions, we define a `Worker` peer. These peers define pairs, which are directly connected. For each pair the peers have the same name that indicates the face/edge/corner and the suffix `Pos` or `Neg` for the orientation. For example, the pair for the communication over faces along the row axis are shown in Listing 4.

Furthermore, for all 27 possible oriented connection combinations, a `Domain` peer is defined. These types are the instantiated worker processes. Listing 5 shows the peer definition for one of the 8 corner domains in the simulation space.

```
@peer type FaceRowPos <: Worker {
  type Tie <: Single[Master] with Single[FaceRowNeg] }
@peer type FaceRowNeg extends Worker {
  type Tie <: Single[Master] with Single[FaceRowPos] }
```

Listing 4. Example neighborhood peers for LULESH.

```
@peer type CornerPlanePosRowPosColPosDomain <:
  FacePlanePos with FaceRowPos with FaceColPos with
  EdgePlanePosRowPos with EdgePlanePosColPos with
  EdgeRowPosColPos with CornerPlanePosRowNegColNeg {
    type Tie <: Single[Master] with Single[FacePlanePos]
    with Single[FaceRowPos] with Single[FaceColPos] with
    Single[EdgePlanePosRowPos] with
    Single[EdgePlanePosColPos] with
    Single[EdRowPosColPos] with
    Single[CornerPlanePosRowPosColPos] }
```

Listing 5. Example of a `Domain` peer for LULESH.

While such explicit definitions require a significant amount of code, they also grant a significant advantage to developers. Since all communication channels and their directions are explicit, communication between adjacent domains can now be implemented without any syntactic overhead. In addition, the communication flow is represented explicitly in the code, helping developers to understand the application logic.

Communication Patterns. For global synchronization after each main loop iteration, the C++ implementation of LULESH uses the synchronous `MPI_Allreduce` function, which collects values from all domain processes before continuation. Our ScalaLoci implementation instead uses reactive events, on which each domain publishes its values, and the master aggregates the worker events to a central event. This is done for example for the global time step evaluation:

```
val elapsedTime: Evt[Double] on Worker = // [...]
def mainIteration(): Unit localOn Worker = placed { // [...]
  elapsedTime fire duration
}
val elapsedTimeGlobal: Event[Double] on Master = placed {
  elapsedTime.asLocalFromAllSeq.fold(/* [...] */) { /* [...] */ }
}
```

While this code looks in first place more complicated, it decouples the aggregation communication, which might be necessary for future optimizations of the program.

For communication between neighbors, the C++ implementation uses asynchronous MPI message passing functions. However, this requires a lot of dynamic checks, whether a domain has a neighbor in the direction it wants to communicate, because there is no explicit architecture definition. Moreover, the addressing is based on indexes, which is not intuitive, provides no static correctness guarantees and is as such potentially error-prone. The explicit peer architecture in ScalaLoci with peers for each communication direction improves this by addressing directions by intuitive names, which makes dynamic checks for partners obsolete, and by ensuring correctness of the communication pattern statically.

For instance, the nodal mass value of the adjacent neighbor located at the face in the positive row direction is transferred:

```
val nmFaceRowPos: Event[Double] on FaceRowPos = // [...]
placed[FaceRowNeg].local {
  nmFaceRowPos.asLocal observe {
    /* Processing with nodal mass from adjacent domain */ }
}
```

This example shows the decoupled communication and visualizes the intuitive way in which communication partners are addressed in the code. Moreover, through the explicit architecture specification this implied value transfer is executed automatically on all workers, which have such a neighbor, and not deployed to any worker without such a neighbor.

4 Preliminary Performance Evaluation

We evaluate the run time of the C++ LULESH implementation and our ScalaLoci version. All measurements are taken on a c5.metal [34] AWS EC2 cloud computing instance with 96 cores of Intel's Cascade Lake processors at 3.6/3.9 GHz turbo frequencies, and 192 GB of memory.

We considered the execution in several configurations AXY and BXY (Table 1) which are defined as follows. Each configuration defines the number of processes n, the number of OpenMP threads per process o, the total thread count $t = n \cdot o$, the LULESH size parameter s, and the resulting computational problem size $p = n \cdot s^3$.

Figure 3 shows the results for the configurations defined in Table 1; less run time is better. Generally, the ScalaLoci version is slower. For the single process executions, we can see that the ScalaLoci implementation benefits from OpenMP parallelization and is single threaded roughly 4 times slower than the C++ reference.

For the measurements with 8 processes, both implementations have increasing run time with higher number of threads due too OpenMP overhead at smaller per-domain complexity. Moreover, the inter-process communication is less efficient for ScalaLoci.

As major bottlenecks in our ScalaLoci adaption we identified that **TCP based communication** causes significant overhead compared to many MPI implementations, for example shared-memory communication in this case study. In contrast to TCP, MPI also enables the usage of faster networking standards like Infiniband in computing clusters. To make use of these techniques, a MPI based communication backend for ScalaLoci is planed. **Immutable objects** for the domain data structure are changed frequently and cause a lot of short living objects. In this LULESH case study these objects cause expensive overhead, because the size of the domain causes the short lived objects to move out of Eden heap space before they are released again. Therefore, garbage collection is very expensive. This can be solved with mutable data structures. **JNI calls** introduce a small time overhead.

Configuration	A01	A02	A08	B01	B04	B12
Processes n	1	1	1	8	8	8
OMP Threads o	1	2	8	1	4	12
Total Threads t	1	2	8	8	32	96
Size Parameter s	20	20	20	10	10	10
Problem Size p	8 k	8 k	8 k	8 k	8 k	8 k

Table 1. Run time evaluation configurations.

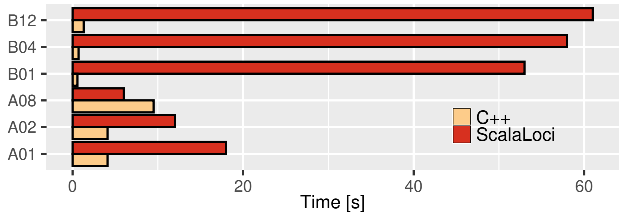

Figure 3. Run time results for LULESH.

Bundling of JNI calls and reduction of overhead will be considered. Alternatively, LLVM based techniques could further reduce this overhead. The **data exchange** between C++ and ScalaLoci is currently inefficient, because data is serialized and copied in memory for each JNI call. Shared non-manged memory buffers outside the heap could be accessed directly by both techniques and reduce this overhead.

5 Outlook

The observations we collected through the LULESH case study indicate our directions for future work.

Architectures. The LULESH case study adopts a well defined architecture specification, as it is the case for HPC applications which need to refer to the configuration of the processors available for data processing. Future work shall explore how code overhead to express complex architectures can be reduced while maintaining the same static guarantees, and how semi-dynamic architecture specifications could help with this approach.

Communication Patterns. The experience with gained in the development of the LULESH benchmark suggests that a set of reusable distributed communication patterns should be made available. We plan to work on this direction by extending ScalaLoci. These shall be solutions for load-balancing, scalability and fault-tolerance for simplified development of high quality HPC algorithms. Plugin implementations of these patterns could be provided as ScalaLoci modules.

Automated Placement. Explicit placement like in ScalaLoci offers the highest level of safety, but might hinder performance optimizations based on automatic runtime placement of computations. We plan to explore controlled and automated placement of values and functions – statically at compile time as well as dynamically at run time. Both can also be leveraged for improvements on load-balancing, scalability and fault-tolerance.

Performance Improvements. Language techniques like virtual methods dispatching or GC are common in managed languages but introduce significant overhead. Yet, we strongly believe their benefits should be accessible to HPC developers. We want to combine our approach with native code that is then integrated into the ScalaLoci programming model. Such native functions could be generated automatically, e.g., with Scala Native, or they could be provided by the programmer.

6 Related Work

We discuss related work regarding PGAS languages, multitier programming and reactive programming.

PGAS/APGAS Languages. The Partitioned Global Address Space (PGAS) model is a parallel programming model similar to both the shared memory and the message passing communication models. While cooperating processes may not physically use the same memory space, they can access data that resides on other processes – the memory space appears shared. Remote accesses usually require a special syntax, but, in contrast to message passing, the actual communication to access data from a remote memory space is transparent for programmers. The survey by De Wael et al. [10] classifies PGAS programming languages.

X10 [5] is an object-oriented programming language from IBM Research aiming to increase programmer productivity in HPC. X10 features distribution of arrays over so-called *places*, abstractions that define a partitioning of the global address space. For parallel programming, X10 introduces *asynchronous activities* that can be spawned individually or by using parallel loops. Activities are associated with a place. Hence remote data can be accessed by spawning on a place an activity which reads and returns the value or modifies it.

Chapel [4] has a similar data distribution model and supports both data and task parallelism. The language provides a more general way of expressing distribution: Besides arrays, other data structures, like sets or graphs, can be distributed.

In addition to distributed data structures, Fortress [1] features constructs for explicit and implicit parallelism. To bring programs closer to their mathematical counterpart and thus increase productivity in scientific computing, the Fortress syntax is designed to adopt unicode characters (e.g., the sum operator Σ).

Multitier Languages. Multitier languages aim to remove the separation of code between communicating processes. In the Web context, this is achieved by compiling the client code to JavaScript or by using JavaScript on the server, too. In Links [7] and Opa [28], function annotations specify whether a function is executed on the client or on the server. Similarly, in StiP.js [25], annotations assign code fragments to the client or the server. In addition, a program slicing mechanism detects dependencies between fragments and the rest of the program and decides where functionalities are placed.

Ur/Web [6] is a multitier language for the Web similar to Haskell. Eliom is a multitier extension of ML [26, 27]. Both assign code placement based on user annotations and check the correctness of placements in the compiler.

The approaches discussed so far focus on the Web. Instead, ML5 [23] is a multitier language for *generic* software architectures. Similarly, ScalaLoci is a Scala extension that supports distributed systems whose architecture is specified by the user via an architectural specification. The compiler then checks that the application complies with the specified architecture.

Functional Reactive Programming. FRP [12] allows the definition of data flows declaratively and concisely via discrete time-changing values (events) and continuous time-changing values (signals). The runtime of reactive languages automatically propagates the changes to dependent values, which usually form a dependency graph. Over time a number of variants have been proposed, with different strategies for the change propagation over the graph. Bainomugisha et al. [2] provide an overview of the existing solutions.

Flapjax [20] introduced the use of FRP in Web applications, which has recently inspired a number of reactive libraries such as Rx.JS [19] and Bacon.js [24]. REScala [29] integrates event streams and time-changing values with object-oriented abstractions. Its runtime provides an event propagation system supporting dynamic dependencies for adding event queries during the execution.

Recent research on RP focused on different propagation strategies to achieve properties such as glitch avoidance [8], efficient propagation over remote network connections [11] or concurrent propagation in a Web environment [9].

7 Conclusion

In this in paper, we present our ongoing approach to increase the level of abstraction in HPC programming. We propose to apply multitier programming and reactive programming techniques to access complex processor configurations uniformly and to specify their communication patterns. Our work bases on ScalaLoci, which we use for a first case study based on the LULESH benchmark, and shows that this research has the potential to improve the software design of HPC applications.

Acknowledgments

This work has been supported by the German Research Foundation (DFG) within the Collaborative Research Center (CRC) 1053 MAKI and 1119 CROSSING, by the DFG projects SA 2918/2-1 and SA 2918/3-1, by the Hessian LOEWE initiative within the Software-Factory 4.0 project, by the German Federal Ministry of Education and Research and by the Hessian Ministry of Science and the Arts within CRISP, and by the AWS Cloud Credits for Research program.

References

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele, and Sam Tobin-Hochstadt. 2008. *The Fortress Language Specification.* Technical Report. Sun Microsystems, Inc. 262 pages. Version 1.0.

[2] Engineer Bainomugisha, Andoni Lombide Carreton, Tom van Cutsem, Stijn Mostinckx, and Wolfgang de Meuter. 2013. A Survey on Reactive Programming. *Comput. Surveys* 45, 4, Article 52 (Aug. 2013), 34 pages. https://doi.org/10.1145/2501654.2501666

[3] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. 2011. Habanero-Java: The New Adventures of Old X10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java (PPPJ '11).* ACM, New York, NY, USA, 51–61. https://doi.org/10.1145/2093157.2093165

[4] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.* 21, 3 (Aug. 2007), 291–312. https://doi.org/10.1177/1094342007078442

[5] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.* 40, 10 (Oct. 2005), 519–538. https://doi.org/10.1145/1103845.1094852

[6] Adam Chlipala. 2015. Ur/Web: A Simple Model for Programming the Web. In *Proceedings of the 42nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '15).* ACM, New York, NY, USA.

[7] Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. 2007. Links: Web Programming Without Tiers. In *Proceedings of the 5th International Conference on Formal Methods for Components and Objects (FMCO '06).* Springer-Verlag, Berlin, Heidelberg.

[8] Gregory H Cooper and Shriram Krishnamurthi. 2006. Embedding dynamic dataflow in a call-by-value language. In *European Symposium on Programming.* Springer, 294–308.

[9] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13).* ACM, New York, NY, USA, 411–422. https://doi.org/10.1145/2491956.2462161

[10] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. 2015. Partitioned Global Address Space Languages. *ACM Comput. Surv.* 47, 4, Article 62 (May 2015), 27 pages. https://doi.org/10.1145/2716320

[11] Joscha Drechsler, Guido Salvaneschi, Ragnar Mogk, and Mira Mezini. 2014. Distributed REScala: An Update Algorithm for Distributed Reactive Programming. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA '14).* ACM, New York, NY, USA, 361–376. https://doi.org/10.1145/2660193.2660240

[12] Conal Elliott and Paul Hudak. 1997. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming (ICFP '97).* ACM, New York, NY, USA, 263–273. https://doi.org/10.1145/258948.258973

[13] Message P Forum. 1994. MPI: A Message-Passing Interface Standard.

[14] Al Geist and Daniel A Reed. 2017. A Survey of High-performance Computing Scaling Challenges. *Int. J. High Perform. Comput. Appl.* 31, 1 (Jan. 2017), 104–113. https://doi.org/10.1177/1094342015597083

[15] David Grove, Sara S. Hamouda, Benjamin Herta, Arun Iyengar, Kiyokuni Kawachiya, Josh Milthorpe, Vijay Saraswat, Avraham Shinnar, Mikio Takeuchi, and Olivier Tardieu. 2019. Failure Recovery in Resilient X10. *ACM Trans. Program. Lang. Syst.* 41, 3 (July 2019), 15:1–15:30. https://doi.org/10.1145/3332372

[16] Sara S. Hamouda, Benjamin Herta, Josh Milthorpe, David Grove, and Olivier Tardieu. 2016. Resilient X10 over MPI User Level Failure Mitigation. In *Proceedings of the 6th ACM SIGPLAN Workshop on X10 (X10 2016).* ACM, New York, NY, USA, 18–23. https://doi.org/10.1145/2931028.2931030 event-place: Santa Barbara, CA, USA.

[17] R. D. Hornung, J. A. Kaesler, and M. B. Gokhale. 2011. *Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory.* Technical Report LLNL-TR-490254. 1–17 pages.

[18] Ian Karlin, Jeff Keasler, and Rob Neely. 2013. *LULESH 2.0 Updates and Changes.* Technical Report LLNL-TR-641973. 1–9 pages.

[19] Erik Meijer. 2010. Reactive Extensions (Rx): Curing Your Asynchronous Programming Blues. In *ACM SIGPLAN Commercial Users of Functional Programming (CUFP '10).* ACM, New York, NY, USA, Article 11, 1 pages. https://doi.org/10.1145/1900160.1900173

[20] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: a programming language for Ajax applications. In *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications (OOPSLA '09).* ACM, New York, NY, USA, 1–20. https://doi.org/10.1145/1640089.1640091

[21] Josh Milthorpe, David Grove, Benjamin Herta, and Olivier Tardieu. 2015. Exploring the APGAS Programming Model using the LULESH Proxy Application. (2015), 7.

[22] Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. 2018. Fault-tolerant Distributed Reactive Programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs)),* Todd Millstein (Ed.), Vol. 109. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:26. https://doi.org/10.4230/LIPIcs.ECOOP.2018.1

[23] Tom Murphy, VII., Karl Crary, and Robert Harper. 2008. Type-safe Distributed Programming with ML5. In *Proceedings of the 3rd Conference on Trustworthy Global Computing (TGC '07).* Springer-Verlag, Berlin, Heidelberg.

[24] Juha Paananen. 2012. Bacon.js. http://baconjs.github.io/. Accessed 2019-08-01.

[25] Laure Philips, Coen De Roover, Tom Van Cutsem, and Wolfgang De Meuter. 2014. Towards Tierless Web Development Without Tierless Languages. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software (Onward! 2014).* ACM, New York, NY, USA.

[26] Gabriel Radanne and Jérôme Vouillon. 2017. Tierless Modules. (March 2017). https://hal.archives-ouvertes.fr/hal-01485362 working paper or preprint.

[27] Gabriel Radanne and Jérôme Vouillon. 2018. Tierless Web Programming in the Large. In *Companion Proceedings of the The Web Conference 2018 (WWW '18).* International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, Switzerland.

[28] David Rajchenbach-Teller and Franois-Régis Sinot. 2010. Opa: Language support for a sane, safe and secure web. *Proceedings of the OWASP AppSec Research* 2010, 1 (2010).

[29] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY '14).* ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2577080.2577083

[30] Guido Salvaneschi and Mira Mezini. 2016. Debugging for Reactive Programming. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16).* ACM, New York, NY, USA.

[31] G. Salvaneschi, S. Proksch, S. Amann, S. Nadi, and M. Mezini. 2017. On the Positive Effect of Reactive Programming on Software Comprehension: An Empirical Study. *IEEE Transactions on Software Engineering (TSE)* PP, 99 (2017), 1–1. https://doi.org/10.1109/TSE.2017.2655524

[32] Manuel Serrano, Erick Gallesio, and Florian Loitsch. 2006. Hop: A Language for Programming the Web 2.0. In *Companion to the 21th ACM SIGPLAN Conference on Object-Oriented Programming, Systems,*

Languages, and Applications (Companion to OOPSLA '06). ACM, New York, NY, USA.

[33] Manuel Serrano and Vincent Prunet. 2016. A Glimpse of Hopjs. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP '16)*. ACM, New York, NY, USA.

[34] Amazon Web Services. 2019. Amazon EC2 C5 Instances. Retrieved 2019-08-14 from https://aws.amazon.com/ec2/instance-types/c5

[35] Pascal Weisenburger, Mirko Köhler, and Guido Salvaneschi. 2018. Distributed System Development with ScalaLoci. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 129 (Oct. 2018), 30 pages. https://doi.org/10.1145/3276499

[36] Pascal Weisenburger and Guido Salvaneschi. 2019. Multitier Modules. In *33rd European Conference on Object-Oriented Programming (ECOOP 2019) (Leibniz International Proceedings in Informatics (LIPIcs))*, Alastair F. Donaldson (Ed.), Vol. 134. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 3:1–3:29. https://doi.org/10.4230/LIPIcs.ECOOP.2019.3