RELEASE: A High-level Paradigm for Reliable Large-scale Server Software

Olivier Boudeville, Francesco Cesarini, Natalia Chechina,
Kenneth Lundin, Nikolaos Papaspyrou, Konstantinos Sagonas,
Simon Thompson, Phil Trinder, and Ulf Wiger

[1] Heriot-Watt University, Edinburgh, EH14 4AS, UK
[2] University of Kent, Canterbury, CT2 7NF, UK
[3] Erlang Solutions AB, 113 59 Stockholm, Sweden
[4] Ericsson AB, 164 83 Stockholm, Sweden
[5] Institute of Communication and Computer Systems (ICCS), Athens, Greece
[6] EDF R&D, 92149 Clamart, France
[7] Uppsala University, 751 05 Uppsala, Sweden

Abstract. Erlang provides a fault-tolerant, reliable model for building concurrent, distributed system based on functional programming. In the RELEASE project the Erlang model is extended to Scalable Distributed Erlang – SD Erlang – supporting general-purpose computation in massively multicore systems. This paper outlines the RELEASE proposal, and indicates the progress of the project in its first six months.

1 Introduction

There is a widening gap between state of the art in hardware and software. Architectures are inexorably becoming multicore, with the numbers of cores per chip following Moore's Law. Software written using conventional programming languages, on the other hand, is still essentially sequential: with a substantial effort, some degree of concurrency may be possible, but this approach just doesn't scale to 100s or 1000s of cores. However, multicore programming is not only about concurrency. Many expect 100,000 core clouds/platforms to become commonplace, and the best predictions are that core failures on such an architecture will become relatively common, perhaps one hour mean time between core failures. So multicore systems need to be both scalable and robust.

The project aim is to scale the radical concurrency-oriented programming paradigm to build reliable general-purpose software, such as server-based systems, on massively parallel machines. Concurrency-oriented programming is distinctive as it is based on highly-scalable lightweight processes *which share nothing*.

The trend-setting concurrency-oriented programming model we will use is Erlang/OTP – designed in the telecoms sector, and used in strategic Ericsson products such as the AXD301 telecoms switch. Erlang/OTP provides high-level coordination with concurrency and robustness built-in: it can readily support 10,000 processes per core, and transparent distribution of processes across multiple machines, using message passing for communication. Moreover, the robustness of the Erlang distribution model is provided

by hierarchies of supervision processes which manage recovery from software or hardware errors.

Currently Erlang/OTP has inherently scalable computation and reliability models, but in practice scalability is constrained by the transitive sharing of connections between all nodes and by explicit process placement. Moreover programmers need support to engineer applications at this scale and existing profiling and debugging tools don't scale, primarily due to the volumes of trace data generated. The RELEASE consortium is uniquely qualified to tackle these challenges working at three levels:

1. *evolving the Erlang virtual machine* so that it can work effectively on large scale multicore systems;
2. *evolving the language to Scalable Distributed (SD) Erlang*, and adapting the OTP framework to provide both constructs like locality control, and reusable coordination patterns to allow SD Erlang to effectively describe computations on large platforms, while preserving performance portability;
3. *developing a scalable Erlang infrastructure* to integrate multiple, heterogeneous clusters.

These developments will be supported by state of the art tools which will allow programmers to understand the behaviour of large scale SD Erlang programs, and to refactor standard Erlang programs into SD Erlang. We will demonstrate the effectiveness of the RELEASE approach through building two significant demonstrators: a simulation based on a port of SD Erlang to the Blue Gene architecture and a Large-Scale Continuous Integration Service, and by investigating how to apply the model to an Actor framework for a mainstream language.

The Erlang community is growing exponentially, moreover it has defined concurrency-oriented programming and become a beacon language for reliable distributed computing. As such it influences the design and implementation of numerous Actor programming languages, libraries and frameworks, like Scala, F# and the Kilim framework for Java. Hence we expect that RELEASE will have a similar impact on the design and implementation of a range of languages, libraries and frameworks, and thus deliver significant results beyond the Erlang community.

The paper is organised as follows. The background information and project goals beyond the state-of-the-art are presented in Section 2. The details of the project plans and means of achieving them are discussed in Section 3. Case studies and benchmarks are covered in Section 4. Finally, the project to date progress and the paper summary are presented in Sections 5 and 6 respectively.

2 Progress Beyond the State-of-the-art

The project makes advances in a number of areas, primarily in scalable high-level distributed languages, in Virtual Machine (VM) support for high-level concurrency, in tools for concurrent software engineering, in cloud hosting infrastructure and in massive scale simulation.

2.1 Heterogeneous General Purpose Computing

We target reliable scalable general purpose computing on heterogeneous platforms. Our application area is that of general server-side computation, e.g. a web or messaging server. This form of computation is ubiquitous, in contrast to more specialised forms such as traditional high-performance computing. Moreover, this is computation on stock platforms, with standard hardware, operating systems and middleware, rather than on more specialised platforms on specific hardware.

We are not targeting capability level HPC, with specialist expensive hardware, and rather specialised applications, e.g. manipulating matrices of floating point numbers. Rather than targeting exascale computing on 10^6 cores [27] we aim for 10^5 cores. For example, the Blue Gene/P that will be exploited during the project has 65,000 cores. Our focus on commodity hardware implies that we do not aim to exploit the experimental many-core architectures like the Intel Terascale.

2.2 Scalable Reliable Programming Models

Shared memory concurrent programming models like OpenMP [5] or Java Threads are generally simple and high level, but don't scale well beyond 10^2 cores. Moreover reliability mechanisms are greatly hampered by the shared state, for example a lock becomes permanently unavailable if the thread holding it fails. In the HPC environment the distributed memory model provided by the MPI communication library [23] dominates. Unfortunately MPI is not suitable for producing general purpose concurrent software as it is too low level with explicit, synchronous message passing. Moreover the most widely used MPI implementations offer no fault recovery[8]: if any part of the computation fails, the entire computation fails.

For scalable high-level general purpose concurrent programming some more flexible model is required. Actors [9] are a widely used model and are built into languages like Scala [17], Erlang [4], and many others. There are also Actor frameworks or libraries for many languages, for example, Termite Scheme [7], PARLEY for Python [19], and Kilim and jetlang are two of several available for Java [24].

Erlang provides an Actor model and is widely recognised as a beacon language for concurrent or distributed computing. That is, it has influenced the design of the concurrent coordination provided in many languages and frameworks, for example Scala [17], F# [25] and Clojure [10]. Let us briefly summarise the key aspects of Erlang style concurrency [29]: fast process creation/destruction; scalability (to support more than 10^4 concurrent processes); fast asynchronous message passing; copying message-passing semantics (share-nothing concurrency); process monitoring; selective message reception.

Figure 1 illustrates Erlang's support for concurrency, multicores and distribution. A blue rectangle represents a host with an IP address, and a red arc represents a connection between nodes. Multiple Erlang processes may execute in a node, and a node can exploit multiple processors, each having multiple cores. Erlang supports single core concurrency as a core may support as many as 10^8 lightweight processes [1]. In the

[8] Some fault tolerance is provided in less widely used MPI implementations like [6].

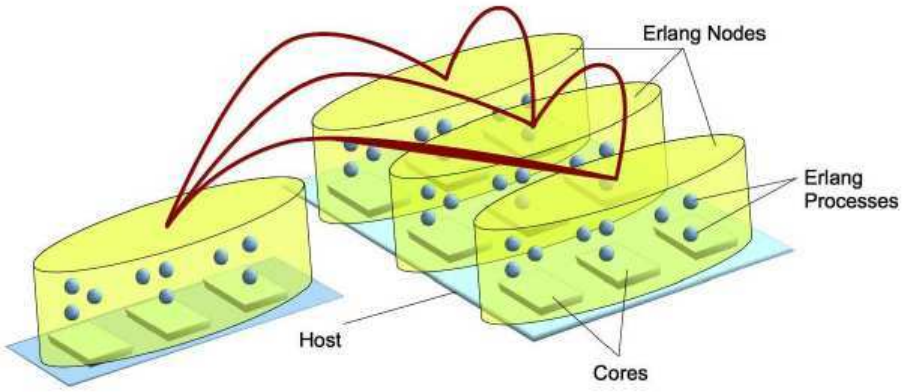

Fig. 1: Conceptual View of Erlang's Concurrency, Multicore Support and Distribution

Erlang distribution model a node may be on a remote host, and this is almost entirely transparent to the processes. Hosts need not be identical, nor do they need to run the same operating system.

Erlang currently delivers reliable medium scale distribution, supporting up to 10^2 cores, or 10^2 distributed memory processors. However, the scalability of a distributed system in Erlang is constrained by the transitive sharing of connections between all nodes and by explicit process placement. The transitive sharing of connections between nodes means that the underlying implementation needs to maintain data structures that are quadratic in the number of processes, rather than considering the communication locality of the processes. While it is possible to explicitly place large numbers of processes in a regular, static way, explicitly placing the irregular or dynamic processes required by many servers and other applications is far more challenging. The project addresses these limitations.

2.3 VM Support for High-level Concurrency

Due to the development of multi-core architectures and the distributed nature of modern computer systems, the quantity and variance of processors on which software is executed has increased by orders of magnitude in recent years and is expected to increase even more. In this setting, as discussed in the previous section, the role of high-level concurrent programming models is very important. Such models must abstract from variations introduced by differences between multi-core architectures and uniformly treat different hardware architectures, number of cores and memory access characteristics. Currently, the implementation of such models in the form of high-level languages and libraries is not sufficient; these two must be complemented with efficient and reliable virtual machines (VMs) that provide inherent support for high-level concurrency [13].

Historically the efficient implementation of VMs for single core processor systems has presented a number of challenges, largely unrelated to concurrency. For example, in order to optimally use the hardware, a VM has to exploit deep hierarchies of cache memory by reorganizing data layouts, and to support e.g. out-of-order execution, the hardware's prefetching heuristics, branch prediction. With multi-core processors, concurrency and the inherent shared memory model introduce new challenges for VMs, as it is not only arbitrary thread interleavings but parallel execution on limited shared resources which has to be taken into account. On top of the implementation considerations mentioned before, cache coherence becomes a critical issue, memory barriers become a necessity and compiler optimizations (such as instruction reordering) must often be more conservative to be semantics-preserving. Furthermore, multi-core machines currently rely on non-uniform memory access (NUMA) architectures, where the cost of accessing a specific memory location can be different from core to core and data locality plays a crucial role for achieving good performance.

In the past few years, there has been sustained research on the development of VMs for software distributed shared memory. Some of the more recent such research aims to effectively employ powerful dedicated and specialized co-processors like graphic cards. Notable VM designs in this direction are the CellVM [15], [30] for the Cell Broadband Engine, a VM with a distributed Java heap on a homogeneous TILE-64 system [28]; an extension of the JikesVM to detect and offload loops on CUDA devices [12]; and VMs for Intel's Larrabee GPGPU architecture [22]. Most of these designs are specific to VMs for Java-like languages which are based on a shared-memory concurrent programming model. Despite much research, the implementation of shared memory concurrency still requires extensive synchronization and for this reason is inherently non-scalable. For example, in languages with automatic memory management, the presence of a garbage collector for a heap which is shared among all processes/threads imposes a point of synchronization between processes and thus becomes a major bottleneck.

Although a language based on the Actor concurrency model is in principle better in this respect, its VM implementation on top of on shared memory hardware in an efficient and scalable way presents many challenges [11]. In the case of the implementation of Erlang, various runtime system architectures have been explored by the HiPE (High Performance Erlang) group in Uppsala, based either on process-local memory areas, on a communal heap which is shared among all threads, or following some hybrid scheme. However, the performance evaluation [21] was conducted on relatively small multi processor machines (up to 4 CPUs) in 2006 and cannot be considered conclusive as far as scalability on the machines that the RELEASE project is aiming at. More generally, the technology required to reach the scalability target of the current project requires significant extensions to the state of the art in the design of virtual machines for message passing concurrency and will stretch the limits of data structures and algorithms for distributed memory management.

2.4 Tools for Concurrency and Erlang

From the inception of parallel programming, the availability of tools that ease the effective deployment of programs on parallel platforms has been a crucial criterion for success. The Intel Trace Analyzer and Collector [14] is a typical modern tool, which sup-

ports the analysis, optimisation and deployment of applications on Intel-based clusters with MPI communication.

The Erlang Virtual Machine is equipped with a comprehensive, low-level tracing infrastructure provided by the trace built-in functions [4]. Built upon this are a number of higher-level facilities, including the debugger (dbg) and the trace-tool builder (TTB) [18], which provides facilities for managing tracing across a set of distributed Erlang nodes. As the name suggests, the tool is designed to be extensible to provide different types of tracing tuned to different applications and environments. While these tools support monitoring Erlang programs across distributed nodes, there is a problem in dealing with the volume of data generated; on a multicore chip, or highly parallel system, it will be impossible to ship all the data off the chip without swamping communication entirely [31]. So, we will ensure that local monitoring and analysis can be performed within the system, leveraging the locality supported by SD Erlang to support a hierarchical 'tracing architecture'.

In building analyses we will be able to leverage work coming from the FP7 ProTest project [20] including the online monitoring tool Inviso, which has recently been integrated into TTB, and the offline monitoring tool Exago. Higher-level analysis can be provided independently [3], and building on this it is also possible to analyse the process structure of Erlang applications, particularly those structured to use the OTP standard behaviours [16, Chapter 6]. These existing systems need to be extended to integrate monitoring with process discovery.

Building on the Erlang syntax_tools package and the standard Erlang compiler tool chain, Wrangler is a tool for refactoring Erlang programs [2]. The Erlang group at Elte, Budapest has built a similar tool, RefactorErl, based on their own compiler front-end infrastructure. In the RELEASE project we will develop and implement refactorings from Erlang to SD Erlang within Wrangler. In order to give users guidance about which refactorings could be applied we will develop a refactoring assistant, which will suggest refactorings of a given system on the basis of its behaviour of a system, given by the tracing tools.

Traditional breakpoint debuggers are of little use for RMP software; there are too many processes and breakpoints conflict with the frequent timeouts in Erlang's communication. The RELEASE project will build a debugger for massively parallel systems that retains a recent history of the current computation state, as well as saving fault-related information, so that this can be retraced and explored for debugging purposes.

2.5 Cloud Hosting

The cloud hosting area is moving rapidly and progress is driven mainly by entrepreneurs and competing cloud infrastructure providers. While users can already choose between many hosting providers offering price/performance alternatives, provisioning is typically manual and time consuming, making it unnecessarily difficult to switch providers.

With increased competition comes specialisation, which in its turn introduces an integration challenge for users. This calls for a broker layer between users and cloud providers [8], but creating such a broker layer, especially a dynamic one, is no easy task, not least because Cloud Provision APIs currently do not support ad-hoc, capability-based provisioning and coordination of cloud resources. A problem is that basic cloud

APIs must primarily serve mainstream languages, not least REST (stateless) clients, where management of dynamic and complex state is considered extremely difficult [26].

We intend to advance the concept of a capability-based dynamic cloud broker layer, building on the cloud broker component of Erlang Solution's recently launched Hosted Continuous Integration tool SWARM. SWARM is capable of allocating a mix of cloud and physical resources on demand and running complex tests in parallel, and draws on the ease with which Erlang can spawn and coordinate parallel activities, as well as its considerable strengths as a test automation and load testing environment. We intend to use SWARM itself as a testbed for the concepts in the RELEASE project, aiming to increase both the capabilities of SWARM and its cost-effectiveness.

At a basic level, this broker would be useful for any kind of cloud application (even non-Erlang) making it easier for users to switch provider, and create on-demand, capability-driven mashups of specialised clusters. For more dynamic capabilities, we will lead by providing a Cloud Broker API customised for the Erlang-style programming model.

2.6 Scalable Simulation

Many business and scientific fields would benefit from larger-scale simulations as the need for intrinsically risky extrapolation can be removed if we are able to simulate a complex system as a whole. Moreover, some important behaviours only appear in detailed simulations.

Parallelism poses a number of challenges for simulation, especially as distributed memory clusters are the cost effective scalable platform. Simulations are typically initially developed using sequential, imperative technologies, and these are ill suited for distributed execution. However there is an increasing belief that declarative programming could be more appropriate to harness parallel platforms.

Sim-Diasca is a distributed engine for large scale discrete simulations implemented in Erlang. It is among the most scalable discrete simulation engines and currently able to handle more than one million relatively complex model instances using only hundreds of cores. However more accurate models are required - ideally we would like 50-200 million model instances. We would also like to introduce reliability. To achieve these goals will require scalable reliability improvements across the software stack: VM, language, and engine.

3 Developing a Scalable Programming Model

This section gives an overview of the technical and research components of the project.

3.1 Scaling the Erlang Programming Model

We will extend both Erlang, to produce Scalable Distributed (SD) Erlang, and also the associated OTP library. The SD Erlang name is used only as a convenient means of identifying the language and VM extensions as we expect them to become standard Erlang in future Erlang/OTP releases.

Controlling Connections. The scalability of a distributed Erlang system is constrained by the transitive sharing of connections between all nodes and by explicit process placement. SD Erlang will regain scalability using layering, and by controlling connection locality by grouping nodes and by controlling process placement affinity.

Process Placement. Currently the Erlang distribution model permits explicit process placement: a process is spawned on a named node. Such a static, directive mechanism is hard for programmers to manage for anything other than small scale, or very regular process networks. We propose to add an abstraction layer that maintains a tree of node groups, abstractly modelling the underlying architecture. We will provide mechanisms for controlling *Affinity*, i.e. how close process must be located, e.g. two rapidly communicating processes may need to be located in the same node. We will also provide mechanisms for controlling *Distribution*, i.e. how far the process must be from the spawning process. For example, two large computations, such as simulation components, may need to be placed on separate clusters.

Scaling Reliability. Erlang/OTP has world leading language level reliability. The challenge is to maintain this reliability at massive scale. For example, any node with a massive number of connections should be placed in a different node group from its supervisor. A new OTP principle could be to structure systems with the supervision tree preserving this property.

Performance Portability. The abstract computational control mechanism are not strongly related to a specific architecture. As far as possible we intend to construct performance portable programs, and computational patterns, by computing distance metrics for the affinity and distribution metrics. Moreover, locality control enables us to use layering as a principle to structure systems: for example, the control processes for a layer appearing in a different group from the controlled processes. This facilitates performance portability as the top layers can be refactored for the new architecture while preserving the lower layers unchanged.

Scalable and Portable OTP. Some OTP principles and behaviours will need to be extended with new scalable principles, and perhaps to some extent redesigned and refactored to control locality and support layering. The supervisor group discussed above and the control layering are examples of new scalable principles.

3.2 Scaling the Erlang Virtual Machine

The Erlang Virtual Machine (VM) was initially designed as a machine for supporting cooperative concurrent execution of processes ("green threads") on physical machines with a single CPU. Since 2006, the Erlang VM has been extended to support Symmetric MultiProcessing (SMP) in the form that is commonly found these days in multi-core machines. The goal has been to focus on stability while giving incremental performance improvements in each release. This approach has worked well for the important class of high-availability server software running on machines with up to 16 cores, but needs significant extensions to achieve scalability on bigger multi-core machines.

In the RELEASE project we aim to re-design and improve some core aspects of Erlang's VM so that it becomes possible for applications to achieve highly scalable performance on high-end multicore machines of the future with minimal refactoring of existing applications. Our goal is to push a big part of the responsibility for achieving scalability from the application programmer to the VM. In particular, we will investigate architectural changes and alternative implementations of key components of Erlang's VM that currently hinder scalability of applications on large multi-core machines. A key consideration is to design scalable components without sacrificing crucial aspects of the existing architecture such as process isolation, high-reliability, robustness and soft real-time properties.

To best achieve these goals, we will begin our work by a detailed study of the performance and scalability characteristics of a representative set of existing Erlang applications running on the current VM. This will help us both to identify the major scalability bottlenecks and prioritize changes and extensions of the runtime system and of key components of the VM. Independently of the results of this study however, there are some parts of the VM which could definitely benefit from extensions or redesign.

One of them is the Erlang Term Storage (ETS) mechanism. Although Erlang processes do not share any memory at the language level, at the implementation level, the Erlang VM provides built-ins that allow processes to store terms in shared global data structures, called ETS tables, and to destructively modify their contents. Currently, many Erlang applications make extensive use of this mechanism, either directly in their code or indirectly via using in-memory databases that are built upon them such as mnesia. With the current implementation of ETS, when the number of processes gets big, accesses to these tables become serialization points for applications and hinder their scalability. Moreover, due to the nature of the garbage collector which is currently employed by the Erlang VM, accesses to terms in these tables require a physical copy of the term from the table to the heap of the process. We will investigate scalable designs of ETS tables that avoid these performance bottlenecks. In addition we will experiment with runtime system organizations and design language built-ins that avoid the need for copying data from ETS tables to the process-local memory areas when it is provably safe to do so.

In the current Erlang VM, processes allocate the majority of their data in process-local memory areas. Whenever they need to communicate, they must explicitly copy their data from the heap of the sender to that of the receiver. They also need to wait to get rescheduled when execution reaches a receive statement with no appropriate messages in the process mailbox. We will design and investigate scalable runtime system architectures that allow groups of processes to communicate without the need for explicit copying and we will develop the runtime support for processes to temporarily yield to their appropriate senders when reaching a receive statement that would otherwise block. One possible such architecture is a clustered shared heap aided by the presence of language constructs such as fibers that are available in languages like C++.

In general, a scalable design of the Erlang VM needs to be supported both by language extensions but also by static and dynamic analyses for identifying "frequently-communicating processes" and for guiding the schedulers of the VM. A significant effort in this task is not only to design and implement these analyses, but to also integ-

rate them in the development environment in a smooth and seamless way, preferably one which is transparent to the programmer.

No matter how scalable the underlying VM will get, large scale applications will definitely need tool support for identifying bottlenecks in them (Section 3.4). The Erlang VM will be extended with light-weight infrastructure for efficient online profiling and monitoring of applications and for maintaining performance information about them. Such profiling information will also be used by the underlying VM to guide the scheduler and possible feedback-directed optimizations. We will design and implement lightweight infrastructure for profiling applications while these applications are running and for maintaining performance information about them.

Finally, in order to test the scalability of our implementation and to enable a case study we will port the Erlang VM to a massively parallel platform. Our current plan is to use a Blue Gene/P machine available at EDF, which will be used in the large scale simulation study. This plan may change or be extended to include more platforms if we gain access to more powerful such machines during the duration of the project.

3.3 Scalable Virtualisation Infrastructure

This work package will provide a broker layer capable of creating, managing and dynamically scaling super-clusters of smaller heterogeneous clusters, based on capability profile matching.

Given the aim of the RELEASE project to develop a model of participating clusters, it is logical to also explore the possibility of creating super-clusters of on-demand clusters provisioned from competing Cloud providers. This would make it possible to cost-optimise large clusters by matching capability profiles against the requested work, and combining groups of instance types, possibly from different providers, into a larger grid. The complexities of running a compute task across such a cluster generally fall into the categories addressed within the RELEASE project: providing a layer of distribution transparency across cooperating clusters; monitoring neighbouring clusters and recovering from partial failures; and tolerance to latency variations across different network links.

A possible small-scale use for a Cloud cluster broker could be to act as a ”Pricerunner” for on-demand computing resources. For Erlang Solutions, it is a natural extension of their Hosted Continuous Integration and Testing as a Service offerings (SWARM), where the system can match the computing needs of different build-and-test batches against availability and pricing of virtual images from different providers. In the context of Hosted Continuous Integration, this capability can also be used to simulate both server- and client-side, using different capabilities, and possibly different providers, for each. It needs to be easy to configure and automate.

The infrastructure that we construct will be novel as few cloud computing providers today offer a powerful enough API for this task. For this reason, we will build our own virtualization environment, e.g. based on the Eucalyptus Cloud software, which is API-compatible with Amazon EC2, but available as Open Source.

3.4 Tool Support for the Development of RMP Software

This work package addresses the need to build a number of different tools to support the development and deployment of Erlang programs on heterogeneous multicore architectures, including SD Erlang and the super-clusters.

The Erlang programming model provides abundant concurrency, with no theoretical limit to the number of concurrent processes existing – and communicating – at any particular time. In practice, however, there can be problems in the execution of highly concurrent systems on a heterogeneous multicore framework. Two particular difficulties are, first, *balancing of computational load between cores*. Each Erlang process will run on a single core, and a desirable property of the system is that each core is utilised to a similar extent. Secondly, there may be *bottlenecks due to communication*. Each process executes on a single core, but a typical process will communicate with other processes which are placed on other cores, and in a large system this can itself become a bottleneck. Using more cores can ease load balancing, but communication bottlenecks may be alleviated by keeping related processes close together, potentially on the same core, but these two are in tension.

We will supply tools that can measure and visualise performance in a number of different ways. The Erlang VM is equipped with a comprehensive, low-level tracing infrastructure on which other tools can be built; DTrace also provides tracing support on Unix platforms. Because of the volume of data generated, we will need to ensure that local monitoring and analysis are performed, leveraging the locality properties of the running system. The tools will be built in sequence. First we will develop textual reports and graphical visualizations for offline monitoring, secondly, these will be generated as snapshots of a running system, and finally we will build tools to support interactive, real-time online system monitoring.

For SD Erlang to be taken up in practice, it will be essential to provide users with a migration path from Erlang: using the Wrangler refactoring platform we will provide the infrastructure for users to migrate their programs to SD Erlang, and moreover provide decision-support tools suggesting migration routes for particular code. Finally, we will supply tools to debug concurrency bugs (or ”Heisenbugs”) in SD Erlang programs, and to develop an *intelligent debugger* based on saving partial histories of computations.

4 Case Studies

The case studies will demonstrate and validate the new reliable massively parallel tools and methodologies in practice. The major studies using SD Erlang and the scalable infrastructure are large-scale (10^7 model instances) discrete simulations for EDF, and dynamically scalable continuous integration service for Erlang Solutions. Key issues include performance portability, scalability and reliability for large-scale software. We will also investigate the feasibility of impacting dominant programming models by adding our scalable reliability technologies to an Actor framework for a mainstream programming language, such as Java.

EDF Simulation. The goals of this first case study are to enhance the reliability and hence scalability of an existing open-source simulation engine, Sim-Diasca (which stands for "Simulation of Discrete Systems of All Scales"). Sim-Diasca is a general-purpose engine, dedicated to all kinds of distributed discrete simulations, and currently used as the corner stone for a few simulators of smart metering systems. The increased scalability will enable Sim-Diasca to execute simulations at an unprecedented scale by exploiting large-scale HPC resources, clusters or Blue Gene supercomputer (both of which provide a total of 10^5 cores).

The goal is to make the engine able to resist to the crash of up to a pre-determined number of computing nodes (e.g. $k = 3$ unavailable nodes), knowing that the engine is currently designed to halt on error as soon as at least one node is lost in the course of the simulation. This is a twofold task:

- The main part is to add application-level fault tolerance to the engine, so that it is able to store its state based on triggers (e.g. wall clock durations elapsed, simulation intermediate milestone met); that state is spread over all the computed nodes involved, thus a distributed snapshot is to be performed, and the state of a given node must be duplicated to k other node(s) depending on the targeted k-reliability class, either in RAM or, more probably, on a non-volatile memory (e.g. on local files, or on a distributed file system, or on a replicated database); of course this checkpointing is meant to allow for a later restart, on a possibly different resource configuration (which is the general case).
- The second part of the task is to integrate the lower-level mechanisms for reliability provided by SD Erlang: some will be transparent, whereas others will be used as building blocks for the first part of the task.

This distributed snapshot/restart feature will be implemented as follows: on a trigger, e.g. every N simulation ticks all the attributes of all the simulation instances are replicated on k other nodes. This requires that simulation agents, like the time managers, and data-loggers, have means of serialising their state for future re-use. The main difficulty is to do the same for all model instances (a.k.a. actors).

Tolerating the common case of losing "slave" compute node at the leaf of the simulation tree is the basic requirement. More advanced challenges to be investigated include:

- Tolerating the loss of intermediate nodes, or even the root one. For the latter case-consensus and leader election are required, and may be generic reliability mechanisms for Sim-Diasca.
- The dynamic addition of nodes naturally follows from tolerating the loss of nodes.
- Supporting instance migration, for example, for planned down-time or load balancing.

Target Platforms include both large clusters, i.e. the Blue Genes and others, and also multicore cards, like the Tilera ones. The advantage of the latter is ready access and local administration compared with large clusters.

Continuous Integration Service. The second case study will be to integrate the support for on-demand, capability-driven heterogeneous super clouds into Erlang Solutions' Continuous Integration framework, SWARM. We envisage integrating up to 4 Amazon EC2 clusters ranging from small, currently four 1GHz cores, to large, currently hundreds of 3GHz cores. To meet client requirements we will also include in-house clusters, such as bespoke embedded device clusters (currently 20 ARM cores), or clusters of dedicated machines currently not available in virtualised environments.

Erlang Solutions has a dynamic stream of customer projects and exactly which configurations we will test will depend on the clients' use cases, but already, prospective clients of SWARM include massively scalable NoSQL databases, multi-site instant messaging systems and middleware for nation-wide trading infrastructures, and Erlang Solutions is also currently developing ad-hoc networking solutions for mobile devices. We plan to explore the potential of Hosted Continuous Integration in all these areas, and derive from that experience what a common provisioning framework for heterogeneous clusters should look like.

Scalable Reliability for a Mainstream Actor Framework. The goal of this study is to evaluate how effectively the scalable reliable concurrency oriented paradigm we will develop can be applied to mainstream software development. We will investigate the feasibility and limitations of adding SD Erlang scalability constructs to a popular Actor framework for a dominant language. The investigation will focus on an open source framework with an active user base, and ideally European in origin. One such framework that meets these requirements is Kilim for Java, developed in Cambridge [24].

5 Progress So Far

In the first six months of the RELEASE project we have made the following progress.

We have started design SD Erlang by making an overview of architecture trends, possible failures, and Erlang modules that might impinge on Erlang scalability. We have formulated the main design principles, and analysed in-memory and persistent data storage mechanisms. We also have developed an initial SD Erlang design that includes implicit process placement and scalable groups to reduce node connections.

We have started to benchmark and trace multicore systems using both the built-in Erlang tracing and DTrace for Unix systems. We are also compiling a set of programs for Erlang and in particular Distributed Erlang, so that we can benchmark the performance of multicore SD Erlang on a set of practical problems.

We have produced a survey of the state-of-the-art cloud providers and management infrastructures. The survey is a foundation for the initial work on providing access to cloud infrastructure for different tasks, such as system building, testing and deployment. We have added a few useful features to Sim-Diasca to ease troubleshooting, e.g. a distributed instance tracker. We are also working on a scalable simulation case that can be used for benchmarking purposes.

6 Conclusion

SD Erlang is designed to provide a successful platform for multicore systems as Erlang currently does for concurrent, distributed systems.The Erlang model supports concurrency and fault-tolerance. Given the future trajectory of multicore systems, it is precisely these features that are needed to support massively multicore systems in which the mean time between core failures will be as low as one hour. We look forward to reporting on the progress of the project in the years to come.

7 Acknowledgements

We would like to thank all our colleagues who work on the RELEASE project. This work has been supported by the European Union grant RII3-CT-2005-026133 'SCIEnce: Symbolic Computing Infrastructure in Europe', IST-2011-287510 'RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software', and by the UK's Engineering and Physical Sciences Research Council grant EP/G055181/1 'HPC-GAP: High Performance Computational Algebra and Discrete Mathematics'.

References

1. *Erlang/OTP Efficiency Guide, System Limits*, 2011. http://erlang.org/doc/efficiency_guide/advanced.html#id67011.
2. Wrangler home page, 2011. http://www.cs.kent.ac.uk/projects/wrangler/.
3. T. Arts and L.-A. Fredlund. Trace analysis of erlang programs. *SIGPLAN Not.*, 37(12):18–24, 2002.
4. F. Cesarini and S. Thompson. *Erlang Programming*. O'Reilly, 2009.
5. R. Chandra, L. Dagum, D. Kohr, D. Maydan, J. McDonald, and R. Menon. *Parallel programming in OpenMP*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.
6. D. Dewolfs, J. Broeckhove, V. Sunderam, and G. E. Fagg. FT-MPI, fault-tolerant metacomputing and generic name services: a case study. In *EuroPVM/MPI'06*, pages 133–140, Berlin, Heidelberg, 2006. Springer-Verlag.
7. G. Germain. Concurrency oriented programming in termite scheme. In *ERLANG'06*, pages 20–20, New York, NY, USA, 2006. ACM.
8. G. Group. Gartner says cloud consumers need brokerages to unlock the potential of cloud services, 2009. http://www.gartner.com/it/page.jsp?id=1064712.
9. C. Hewitt, P. Bishop, and R. Steiger. A universal modular actor formalism for artificial intelligence. In *IJCAI'73*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.
10. R. Hickey. The Clojure programming language. In *DLS'08*, pages 1:1–1:1, New York, NY, USA, 2008. ACM.
11. R. K. Karmani, A. Shali, and G. Agha. Actor frameworks for the jvm platform: a comparative analysis. In *PPPJ '09*, pages 11–20, New York, NY, USA, 2009. ACM.
12. A. Leung, O. Lhoták, and G. Lashari. Automatic parallelization for graphics processing units. In *PPPJ'09*, pages 91–100, New York, NY, USA, 2009. ACM.
13. S. Marr, M. Haupt, S. Timbermont, B. Adams, T. D'Hondt, P. Costanza, and W. D. Meuter. Virtual machine support for many-core architectures: Decoupling abstract from concrete concurrency models. In *PLACES*, pages 63–77, 2009.

14. I. S. Network. Intel trace analyzer and collector, 2011. http://software.intel.com/en-us/articles/intel-trace-analyzer/.
15. A. Noll, A. Gal, and M. Franz. Cellvm: A homogeneous virtual machine runtime system for a heterogeneous single-chip multiprocessor. In *Workshop on Cell Systems and Applications*, 2009.
16. J. Nystrom. *Analysing Fault Tolerance for Erlang Application*. PhD thesis, Division of Computer Systems, Uppsala University, 2009.
17. M. Odersky, P. Altherr, and V. Cremet. The scala language specification. Technical report, EFPL, Lausanne, Switzerland, April 2004.
18. E. online documentation. Trace tool builder, 2011. www.erlang.org/doc/apps/observer/ttb_ug.html.
19. PARLEY. Python actor runtime library, 2010. http://osl.cs.uiui.edu/parley.
20. T. P. project. Framework 7 project 215868, 2008-11. www.protest-project.eu.
21. K. Sagonas and J. Wilhelmsson. Efficient memory management for concurrent programs that use message passing. *Sci. Comput. Program.*, 62(2):98–121, 2006.
22. L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):18:1–18:15, 2008.
23. M. Snir, S. W. Otto, D. W. Walker, J. Dongarra, and S. Huss-Lederman. *MPI: The Complete Reference*. MIT Press, Cambridge, MA, USA, 1995.
24. S. Srinivasan and A. Mycroft. Kilim: Isolation-typed actors for java. In *ECOOP'08*, pages 104–128, Berlin, Heidelberg, 2008. Springer-Verlag.
25. D. Syme, A. Granicz, and A. Cisternino. *Expert F#*. Springer, 2007.
26. E. Systems. Cloud apis (blog article), 2010. http://www.eucalyptus.com/blog/2010/03/11/cloud-apis.
27. A. Trew. Parallelism and the exascale challenge. Distinguished Lecture. St Andrews University, 2010.
28. D. Ungar and S. S. Adams. Hosting an object heap on manycore hardware: an exploration. *SIGPLAN Not.*, 44(12):99–110, October 2009.
29. U. Wiger. What is erlang-style concurrency?, 2010. http://ulf.wiger.net/weblog/2008/02/06/what-is-erlang-style-concurrency/.
30. K. Williams, A. Noll, A. Gal, and D. Gregg. Optimization strategies for a java virtual machine interpreter on the cell broadband engine. In *CF'08*, pages 189–198, New York, NY, USA, 2008. ACM.
31. J. Zhao, S. Madduri, R. Vadlamani, W. Burleson, and R. Tessier. A dedicated monitoring infrastructure for multicore processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(6):1011–1022, June 2011.