Session Types: Towards safe and fast reconfigurable programming

Nicholas Ng, Nobuko Yoshida, Xin Yu Niu, Kuen Hung Tsoi, Wayne Luk
Department of Computing,
Imperial College London, UK
{nickng,yoshida,nx210,khtsoi,wl}@doc.ic.ac.uk

ABSTRACT

This paper introduces a new programming framework based on the theory of session types for safe, reconfigurable parallel designs. We apply the session type theory to C and Java programming languages and demonstrate that the session-based languages can offer a clear and tractable framework to describe communications between parallel components and guarantee communication-safety and deadlock-freedom by compile-time type checking. Many representative communication topologies such as a ring or scatter-gather can be programmed and verified in session-based programming languages. Case studies involving N-body simulation and K-means clustering are used to illustrate the session-based programming style and to demonstrate that the session-based languages perform competitively against MPI counterparts in an FPGA-based heterogeneous cluster, as well as the potential of integrating them with FPGA acceleration.

1. INTRODUCTION

The two main ways to improve performance of a program are speeding up serial or parallelising execution of a program. As serial architectures are hitting their limits, the industry has shifted their focus from serial architectures to parallel and heterogeneous architectures, combining parallelism with specialisation of heterogeneous hardware. Sequential programming models and techniques are unsuitable for these parallel architectures as they are, for example, not designed to have access to resources in parallel. As a result, parallel programming techniques such as those using MPI [12] are being developed to understand and make full use of the parallel architectures.

Utilising resources for concurrent execution is, however, far from straightforward – blindly parallelising components with data dependencies might leave the overall program in an inconsistent state; arbitrary interleaving of parallel executions combined with complex flow control can easily lead to unexpected behaviour, such as blocked access to resources in a circular chain (i.e. deadlock) or mismatched send-receive pairs. These unsafe communications are a source of non-termination or incorrect execution of a program. Thus tracking and avoiding communication errors of parallel programs is as important as ensuring their functional correctness.

This paper introduces a new programming framework for reconfigurable parallel designs which can automatically ensure deadlock-freedom and communication-safety, ie. matching communication pairs, based on the theory of *session types* [3, 4]. To illustrate how session types can track communication mismatches, consider the following parallel program that exchanges two values between two processes.

	Process 0		Process 1
t=0	`Recv b`	⇔	`Recv a`
t=1	`Send 42`		`Send 64`

In this notation, the arrow points from the sender of the message to the intended receiver. Both `Process0` and `Process1` start by waiting to receive a value from the other processes, hence we have a typical deadlock situation.

A simple solution is to swap the order of the receive and send commands for one of the processes, for example, `Process0`:

	Process 0		Process 1
t=0	`Send 42`	⟶	`Recv a`
t=1	`Recv b`	⟵	`Send 64`

However, the above program still has mismatched communication pairs and causes the type error. Parallel programming usually involves debugging and resolving these communication problems, which is often a tedious task.

Using the session type-based programming methodology, we can not only statically check that the above programs are incorrect, but can also encourage programmers to write safe designs from the beginning, guided by the information of types. Session types [3, 4] have been actively studied as a high-level abstraction of structured communication-based programming, which are able to accurately and intelligibly represent and capture complex interaction patterns between communicating parties.

As a simple example of session types, recall the above example, which has the following session types:

`Process 0`: Recv `char`; Send `int`
`Process 1`: Recv `char`; Send `int`

where Send `int` stands for output with type `int` and Recv `int` stands for input with type `int`. The session types can be used to check that the communications between `Process 0` and `Process 1` are *incompatible* (i.e. incorrect) because one process must have a *dual type* of the other. Similarly, the second example has the following incorrect session types:

`Process 0`: Send `int`; Recv `char`
`Process 1`: Recv `char`; Send `int`

This work was presented in part at the Third International Workshop on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART2012), Naha, Okinawa, Japan, May 31-June 1, 2012.

On the other hand, the following program is correct, having neither deadlock nor type errors.

	Process 0		Process 1
t=0	`Send b`	⟶	`Recv b`
t=1	`Recv 64`	⟵	`Send 64`

since the above program has the following *mutually dual* session types:

`Process 0`: Send `char`; Recv `int`
`Process 1`: Recv `char`; Send `int`

In session type theory, Recv `type` is dual to Send `type`, hence the type of `Process 0` is dual of the type of `Process 1`.

The above compatibility checking is simple and straightforward in the case of two parties. We can extend this idea to multiparty processes (i.e. more than two processes) based on multiparty session type theory [4]. Type-checking for parallel programs with multiparty processes is done statically and is efficient, with a polynomial-time bound with respect to the size of the program.

Below we list the contributions of this paper.

- Novel programming languages for communications in reconfigurable parallel designs and their validation framework with a guarantee of communication-safety and deadlock-freedom (§ 2);
- Implementations of advanced communication topologies for computer clusters by session types for two applications (§ 3), N-body simulation and K-means clustering in session-based languages on Axel, an FPGA-based heterogeneous cluster (§ 4);
- Performance comparison of the implementations in the session-based C (Session C) and the session-based Java (SJ [5, 6, 9]) against the existingparallel programming language, MPI (OpenMPI) [11] and its Java counterpart (MPJ Express [8]) (§5). The benchmark results on Axel demonstrate that implementations of typical parallel algorithms in Session C and SJ execute competitively against their implementations in OpenMPI and MPJ Express, respectively.

2. NEW LANGUAGE DESIGN

2.1 Overview

As a language independent framework for communication-based programming, session types can be applied to different programming languages and environments. Previous work on Session Java (SJ) [6,9] integrated sessions into the object-oriented programming paradigm as an extension of the Java language, and was applied to parallel programming [9]. Session types have also been implemented in different languages such as OCaml, Haskell, F#, Scala and Python. This section explains session types and their applications, focussing on an implementation of sessions in the C language (Session C) as a parallel programming framework. Amongst all these different incarnations of session types, the key idea remains unchanged. A session-based system provides (1) a set of predefined primitives or interfaces for session communication and (2) a session typing system which can verify, at compile time, that each program conforms to its session type. Once the programs are type checked, they run correctly without deadlock nor communication errors.

2.2 Multiparty session programming

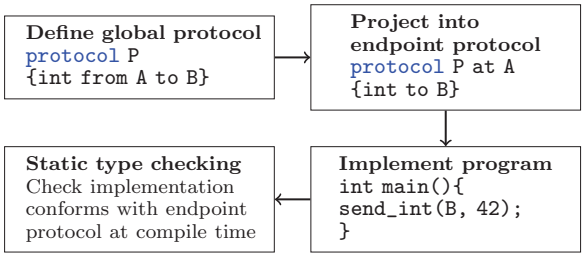

Figure 1: Session C design flow.

Session C [10, 20] implements a generalised session type theory, *multiparty session types* (MPST) [4]. The MPST theory extends the original binary session types [3] by describing communications across multiple participants in the form of *global protocols*. Our development uses a Java-like protocol description language Scribble [2, 13] for describing the multiparty session types. Figure 1 explains a design flow of Session C programming. First, the programmer writes a global protocol starting from the keyword `protocol` and the protocol name. In the first box of Figure 1, the protocol named as `P` contains one communication with a value typed by `int` from participant `A` to participant `B`. For Session C implementation, the programmer uses the *endpoint protocol* generated by the projection algorithm in Scribble. For example, the above global protocol is projected to `A` to obtain `int to B` (as in the second box) and to `B` to obtain `int from A`. Each endpoint protocol gives a template for developing safe code for each participant and as a basis for static verification. Since we started from a correct global protocol, if endpoint programs (in the third box) conform to the induced endpoint protocols, it automatically ensures deadlock-free, well-matched interactions. This endpoint projection approach is particularly useful when many participants are communicating under complex communication topologies. Due to space limitation, this paper omits the full definition of global protocols, and will explain our framework and examples using only endpoint protocols introduced in the next subsection.

2.3 Protocols for session communications

The endpoint protocols include types for basic message-passing and for capturing control flow patterns. We use the endpoint protocol description derived from Scribble to algorithmically specify high-level communication of distributed parallel programs as a library of network communications. A protocol abstracts away the contents but keeps the high level structures of communications as a series of type primitives. The syntax is very compact as given below:

```
<statement> ::= <datatype> "to"
                <participant> ("," <participant>)*
              | <datatype> "from"
                <participant> ("," <participant>)*
              | "choice to" <participant> "{"
                (<label> ":" "{" <statements> "}")+ "}"
              | "choice from" <participant> "{"
                (<label> ":" "{" <statements> "}")+ "}"
              | "rec" <reclabel> "{" <statements> "}"
              | <reclabel>
<statements> ::= <statement> (";" <statement>)*
```

The language above can be categorised to three types of operations: *message-passing*, *choice* and *iteration*.

Message passing. It represents that messages (or data) being communicated from one process to another; in the language it is denoted by the statements `datatype to P1` or `datatype from P0` which stands for sending/receiving data of `datatype` to the participant identified by `P0`/`P1` respectively. Notice that the protocol does not specify the value being sent/received, but instead designate the datatype (which could be primitive types such as `int` or composite types), indicating its nature as a high-level abstraction of communication.

Choice. It allows a communication to exhibit different behavioural flows in a program. We denote a choice by a pair of primitives, `choice from` and `choice to`, meaning a distributed choice receiver and choice maker, respectively. A choice maker first decides a branch to take, identified by its `label`, and executes its associated block of statements. The chosen label is sent to the choice receiver, which looks up the label in its choices and execute the its associated block of statements. This ensures the two processes are synchronised in terms of the choice taken.

Iteration. It can represent repetitive communication patterns. We represent recursion by the `rec` primitive (short for recursion), followed by the block of statements to be repeated, enclosed by braces. The operation does not require communication as it is a local recursion. However two communicating processes have to ensure both of their endpoint protocols contains recursion, otherwise their protocols will not be compatible.

More examples of the endpoint protocols are given in § 3.

2.4 Session C

In Session C, a user implements a parallel program using the API provided by the library, following communication protocols stipulated in Scribble. Once a program is complete, the type checker verifies that the program code matches that of the endpoint protocol description in Scribble to ensure that the program is safe. The core runtime API corresponds the endpoint protocol as described below.

Message passing. Session C's message passing primitives are written as `send_datatype(participant, data)` for message send, which is `datatype to participant` in the protocol, and `recv_datatype(participant, &data)` for message receive (`datatype from participant` in the protocol).

Choice. Session C's choice is a combination of ordinary C control-flow syntax and session primitives. For a choice maker, each if-then or if-else block in a session-typed choice starts with `outbranch(participant, branchLabel)` to mark the beginning of a choice. For a choice receiver, `inbranch(participant, &branchLabel)` is used as the argument of a switch-case statement, and each case-block is distinguished by the `branchLabel` corresponding to a choice in the `choice from` block in the protocol.

Iteration. Session C's iteration corresponds to `while` loops in C. As no communication is required, the implementation simply repeats a block of code consisting of above session primitives in a `rec` recursion block.

A detailed example of Session C will be given in § 4.

3. ADVANCED COMMUNICATION TOPOLOGIES FOR CLUSTERS

This section shows how session endpoint protocols introduced in § 2.3 can be used to specify advanced, complex communications for clusters. Consider a heterogeneous cluster with multiple kinds of acceleration hardware, such as GPUs or FPGAs, as Processing Elements (PEs). To allow a safe and high performance collaborative computation on the cluster, we can describe communications between PEs by our communication primitives. The PEs can be abstracted as small computation functions with a basic interface for data input and result output, hence we can easily describe high-level understanding of the program by the session types.

The following paragraphs list some widely used structured communication patterns that form the backbones of implementations of parallel algorithms. These patterns were chosen because they exemplify representative communication patterns used in clusters. Computation can interleave between statements if no conflict in the data dependencies exists. The implementation follows the theory of the optimisation for session types developed in [7], maximising overlapped messaging. See § 5 for the performance evaluation based on this optimisation.

Ring topology. In a ring topology, as depicted in Figure 2, processes or PEs are arranged in a pipeline, where the end of the node of the pipeline is connected to the initial node. Each of the connections of the nodes is represented by an individual endpoint session. We use N-body simulation as an example for ring topology. Note that the communication patterns between the middle $n-1$ Nodes are identical.

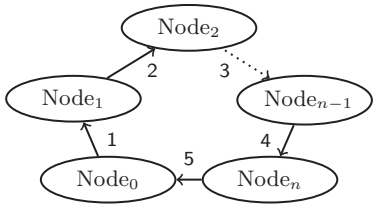

Figure 2: n-node ring pipeline.

The endpoint protocol can precisely represent this ring topology as given below:

```
Node_0:          rec LOOP { // Repeat shifting ring
                   datatype to Node1; // Next node
                   datatype from NodeN; // Last node
                 LOOP }
Node_{1≤i≤n−1}:  rec LOOP { // Repeat shifting ring
                   datatype from Nodei-1; // Prev
                   datatype to Nodei+1; // Next
                 LOOP }
Node_n:          rec LOOP { // Repeat shifting ring
                   datatype from Noden-1; // Prev
                   datatype to Node0; // Initial
                 LOOP }
```

Map-reduce pattern. Map-reduce is a common scatter-gather pattern used to parallelise tasks that can be easily partitioned with few dependencies between the partitioned computations. The topology is shown in Figure 3. It combines the map pattern which partitions and distributes data to parallel workers by a Master coordination node, and the reduce pattern which collects and combines completed results from all parallel workers. At the end of a map-reduce, the Master coordination node will have a copy of the fi-

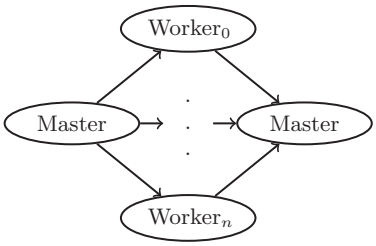

Figure 3: Map-reduce pattern.

nal results combined into a single datum. All Workers in a map-reduce topology share a simple communication pattern, where they only interact with the Master coordination node. The Master node will have a communication pattern that contains all known Workers.

The MPI operation `MPI_Alltoall` is a communication-only instance of the map-reduce pattern for all of the nodes, and only applies memory concatenation to the collected set of data. Our endpoint types given below can represent this topology with more fine-grained primitives so that we can obtain performance gain by communication-computation overlap, see § 5.2.

Master :
```
rec LOOP {
  // Map phase
  datatype to Worker0, Worker1;
  // Reduce phase
  datatype from Worker0, Worker1;
LOOP }
```

Worker$_{0\leq i\leq n}$:
```
rec LOOP {
  // Map phase
  datatype from Master;
  // Reduce phase
  datatype to Master;
LOOP }
```

4. CASE STUDY: K-MEANS CLUSTERING

K-means clustering is an algorithm for grouping a set of objections into k clusters. Initially, k centres of clusters are chosen randomly. Each object will be assigned to a cluster based on their proximity to the nearest centre. After each iteration of the assignment, the centre of the clusters will be recalculated by taking the mean of all objects belonging to that cluster. The whole process will be repeated until the clusters stabilise or reach a pre-determined number of iteration steps. In our implementation, the assignment is parallelised and computed in parallel, and the resulting clusters are distributed between all PEs, so that the centres of the cluster can be calculated on each of the PEs for the next iteration.

Below is the protocol specification of one of our participants, `Worker0`, of our K-means clustering implementation.

```
protocol Kmeans at Worker0 {
  rec STEP {
    // Multicast to Worker1 Worker2 Worker3
    int_array to Worker1, Worker2, Worker3;
    // Multi-receive from Worker1 Worker2 Worker3
    int_array from Worker1, Worker2, Worker3;
  STEP }
}
```

Listing 1: Protocol of the K-means clustering.

The block `rec STEP { }` means recursion, and represents the repeating assign-and-update in the algorithm. The line `int_array to Worker1, Worker2, Worker3` stands for sending from the participant (in this case `Worker0` to `Worker1`, `Worker2`, `Worker3`) with a message of type `int_array`. The implementation of the algorithm in Session C is listed below:

```
for (i=0; i<STEPS; ++i) {
  kmeans_compute_fpga(range_start, range_end);
  // Multicast to Worker1 Worker2 Worker3
  msend_int_array(centres_wkr0, chunk_sz, 3,
                  Worker1, Worker2, Worker3);

  // ... Update centres with local results

  // Multi-receive from Worker1, Worker2, Worker3
  mrecv_int_array(centres_wkr, &sz, 3,
                  Worker1, Worker2, Worker3);

  // ... Update centres with remote results
}
```

Listing 2: Implementation of K-means clustering.

In the code above, `msend_int_array` and `mrecv_int_array` are the variadic primitives for multicast send and multi-receive respectively from the Session C runtime library. The first parameter is the pointer to the data to be sent, followed by the size of the data and the total number of PE identifiers to be sent to or received from. `Worker1`, `Worker2`, `Worker3` are the identifiers of the PEs which this PE is communicating with. The variable `chunk_sz` holds the size of the partition to be `msend` to each participants; and `sz` will contain the total number of bytes received by `mrecv`.

The scatter-gather pattern utilised by the above distributes the local results of partitioned computation to other `Worker`s, then receives the results from them. At the end of the loop, all PEs in the computation will have the complete set of centres from all other PEs. In MPI, this will be a `MPI_Alltoall` operation. On the other hand, the asynchronous communication primitives of Session C allows a partial overlap of communications with the process of updating centres to reduce the execution time. This fine grained control is backed by the session type checking process, ensuring the communications with a partial overlap are deadlock-free.

5. EVALUATION

We evaluate our approach by implementing parallel benchmarks on a heterogeneous cluster, Axel, with the session-based languages, Session C and SJ.

5.1 Hardware environment

Axel [16] is a heterogeneous cluster with 24 nodes. Each of the nodes on the Axel cluster contains a x86 CPU, a number of GPUs and an FPGA board as accelerators. Axel is a Non-uniform Node Uniform Systems (NNUS) cluster system, where each node has a similar configuration consisting of different PEs. CPU, FPGA and GPU are the PEs in each of the nodes in the Axel cluster. Each node can be used as independent x86 PC equipped with hardware accelerators (FPGA board and GPUs), hence new cluster nodes can be built or extended with commodity hardware.

The main communication interface between the nodes is an Ethernet network connected in a star topology between all heterogeneous computing nodes (HCN). Inter-node communication is therefore via a TCP network, by network libraries such as MPI or Session C runtime library. Heteroge-

neous components including the CPU, system memory and FPGA are connected by a common PCIe bus. Data are transmitted to and from the FPGA memory by Direct Memory Access (DMA), and portions of the FPGA memory were mapped to the main memory, providing an easy interface for controlling the FPGA within the node by driver code that runs on the local CPU. Inter-component communication within a node can also be governed by Session C if a separate Session C process is instantiated for FPGA, GPU and CPU, and interact over the loopback network, this also opens up an opportunity for FPGAs to communicate directly over the Infiniband connection.

Although the Axel cluster contains various different kinds of accelerators, they can all be abstracted as homogeneous PEs with a varied amount of computational capabilities, which can all take advantage of the session-based communication provided by the session programming frameworks.

5.2 Benchmark results

We first implemented N-body simulation with Session Java (SJ), a session-enhanced Java [6] and compared our implementation against MPJ Express [8, 14] which is an MPI library in Java. All of our implementations of N-body simulation in SJ/MPJ Express use a ring topology described in §3. SJ shows competitive performance against MPJ Express.

In addition, we configured our implementation to take advantage of the FPGA hardware to accelerate the computation. The execution environment of SJ is the Java Virtual Machine (JVM), and is unable to interface with the FPGA directly, as there are needs to directly access system memory in native environment. Hence Java Native Interface (JNI) was used to bridge the two execution environments.

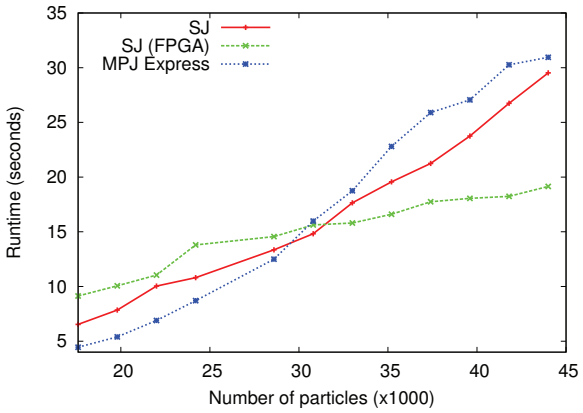

Figure 4: SJ N-body against MPJ Express.

Figure 4 shows the results of the implementations on 11 nodes of the cluster with the same number of steps of each simulation. In addition to the I/O overhead between the software and FPGA, we also take into account of the JNI bridging overhead, which translates JVM data to and from native formats. As the input size increases, the overhead is compensated by the performance of FPGA. The SJ (FPGA) performance is shown on the graph to overtake the non-accelerated SJ implementation at about 32000 particles. This represents the input size when acceleration with FPGA becomes feasible with the given problem.

The results prompted us to further investigate the session-based approach in a pure native high performance heterogeneous computing environment with the Session C framework. Figure 5 compares performance of FPGA-accelerated and non-accelerated Session C and MPI N-body simulation implementations with different input sizes on 6 nodes of the cluster. OpenMPI 1.4 [11] was used as the native MPI implementation, and both Session C and MPI versions share code for computation. We observed similar results as SJ N-body simulation, with the FPGA versions achieving up to 8 times speedup compared to the non-accelerated version. Session C performs almost identically as MPI, showing that the Session C implementation does not add communication overhead to the overall program. We also note that there is much lower FPGA overhead, represented by the intersection point between Session C/MPI (FPGA) and Session C/MPI plots, and hence a significant improvement over previous SJ and Java implementations. Our Session C implementation also utilised less nodes (6 nodes) of the Axel cluster to obtain a superior performance over the SJ implementations (11 nodes).

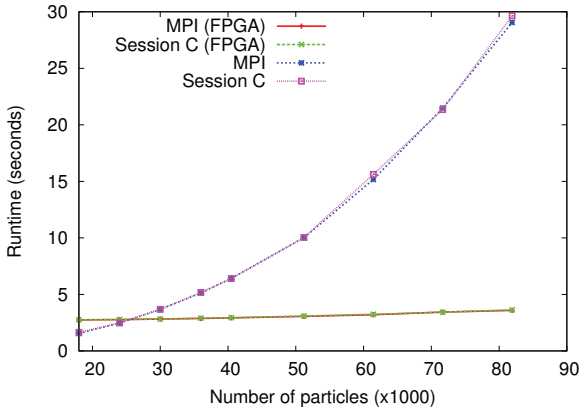

Figure 5: Session C N-body against OpenMPI.

We evaluate the scalability of our implementations by the performance of Session C and MPI parallel N-body simulation and K-means clustering, and the results are shown in Figure 6. The reported runtimes for 1 node is the serial execution of the implementations which is identical in both Session C and MPI version since they share the same code for the main computation. As described in §4, the Session C implementations can take advantage of fine grained controls of communication-overlapping in the K-means clustering algorithm. With more parallel processes, there are more opportunities to overlap computation and communication which results in a diminishing runtime difference between Session C and MPI implementations. It should be noted that although similar optimisation can be applied to MPI using asynchronous communication model, unlike Session C, MPI provides no means to check and ensure the design and transformation is correct in the absence of a session type checker for MPI. Moreover, the session-based approach allows users to verify their optimisations statically by type checking without further testing.

6. SUMMARY

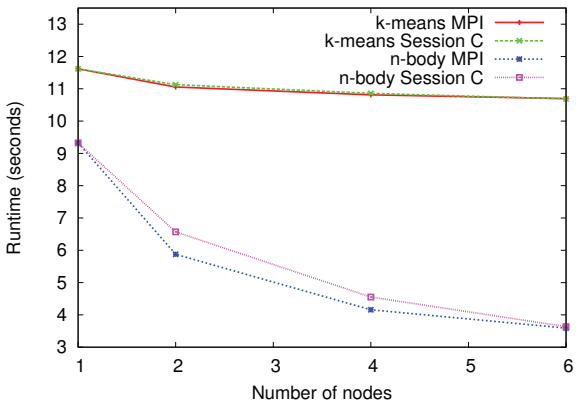

Figure 6: Scalability comparison.

ISP [17] and the distributed DAMPI [18] are formal dynamic verifiers which apply model-checking techniques to standard MPI C source code to detect deadlocks using a test harness. The tool exploits independence between thread actions to reduce the state space of possible thread interleavings of an execution, and checks for potentially violating situations. TASS [15] is another suite of tools for formal verification of MPI-based parallel programs by model-checking. It constructs an abstract model of a given MPI program and uses symbolic execution to evaluate the model, which is checked for a number of safety properties including potential deadlocks and functional equivalences.

Compared to the test-based and model-checking approaches which may not be able to cover all possible states of the model, the session type-based approach does not depend on external testing or extraction of models from program code for safety. It encourages designing communication-correct programs from the start, especially given the high level communication structure which session types captures.

Immediate future work includes extending our session-based, FPGA-enabled approach for safe and collaborative high performance computing with other kinds of acceleration hardware such as GPUs.

The runtime communication library described in this work is an early prototype, which will be further refined for lower latency and higher performance. For example, combining protocol specifications and customisable communication frameworks such as [1] on accelerator equipped systems can result in more compact message formats for inter-node and inter-accelerator communication, hence will achieve better communication performance, especially in the presence of inter-FPGA communication medium.

Developing an MPI-compatible runtime interface is envisaged as a future direction of our session-based C language. This allows us to benefit from state-of-the-art research results on applying MPI as a programming model for high performance reconfigurable system such as [12], or as a software-hardware co-development model such as [19], while having the advantages of session-typed communication programming methodologies including formal safety assurance.

This work is supported by UK EPSRC, Alpha Data, Maxeler, and Xilinx. The research leading to these results has received funding from EPSRC EP/F003757/01, EP/G015635/01 and the European Union Seventh Framework Programme under grant agreements number 248976, 257906 and 287804.

7. REFERENCES

[1] S. Denholm, K. H. Tsoi, P. Pietzuch, and W. Luk. CusComNet: A Customisable Network for Recongurable Heterogeneous Clusters. In *ASAP*, pages 9–16. IEEE, 2011.

[2] K. Honda et al. Scribbling interactions with a formal foundation. In *ICDCIT*, volume 6536 of *LNCS*, pages 55–75. Springer, 2011.

[3] K. Honda, V. T. Vasconcelos, and M. Kubo. Language Primitives and Type Disciplines for Structured Communication-based Programming. In *ESOP*, volume 1381 of *LNCS*, pages 122–138. Springer-Verlag, 1998.

[4] K. Honda, N. Yoshida, and M. Carbone. Multiparty asynchronous session types. In *POPL'08*, volume 5201, page 273, 2008.

[5] R. Hu et al. Type-Safe Eventful Sessions in Java. In *ECOOP*, volume 6183 of *LNCS*, pages 329–353, 2010.

[6] R. Hu, N. Yoshida, and K. Honda. Session-Based Distributed Programming in Java. In *ECOOP*, volume 5142 of *LNCS*, pages 516–541, 2008.

[7] D. Mostrous, N. Yoshida, and K. Honda. Global principal typing in partially commutative asynchronous sessions. In *ESOP*, volume 5502 of *LNCS*, pages 316–332, 2009.

[8] MPJ Express homepage. `http://mpj-express.org/`.

[9] N. Ng et al. Safe Parallel Programming with Session Java. In *COORDINATION*, volume 6721 of *LNCS*, pages 110–126, 2011.

[10] N. Ng, N. Yoshida, and K. Honda. Multiparty Session C: Safe parallel programming with message optimisation. In *TOOLS*, pages 203–219, 2012.

[11] OpenMPI Homepage. `http://www.open-mpi.org/`.

[12] M. Saldaña et al. MPI as a Programming Model for High-Performance Reconfigurable Computers. *TRETS*, 3(4):1–29, Nov. 2010.

[13] Scribble homepage. `http://www.jboss.org/scribble`.

[14] A. Shafi, B. Carpenter, and M. Baker. Nested parallelism for multi-core HPC systems using Java. *JPDC*, 69(6):532–545, June 2009.

[15] S. F. Siegel and T. K. Zirkel. Automatic formal verification of MPI-based parallel programs. In *PPoPP'11*, page 309. ACM Press, Feb. 2011.

[16] K. H. Tsoi and W. Luk. Axel: a heterogeneeous cluster with FPGAs and GFPUs. In *FPGA '10*, pages 115–124. ACM Press, 2010.

[17] A. Vo et al. Formal verification of practical MPI programs. In *PPoPP'09*, pages 261–270, 2009.

[18] A. Vo et al. A Scalable and Distributed Dynamic Formal Verifier for MPI Programs. In *SC'10*, pages 1–10. IEEE, 2010.

[19] J. P. Walters et al. MPI-HMMER-Boost: Distributed FPGA Acceleration. *The Journal of Signal Processing Systems*, 48(3):223–238, Aug. 2007.

[20] Session C homepage. `http://www.doc.ic.ac.uk/~cn06/sessionc/`.