Growing a Software Language for Hardware Design

Joshua Auerbach, David F. Bacon, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla

IBM Thomas J. Watson Research Center
Yorktown Heights, NY, US

Abstract

The Liquid Metal project at IBM Research aimed to design and implement a new programming language called Lime to address some of the challenges posed by heterogeneous systems. Lime is a Java-compatible programming language with features designed to facilitate high level synthesis to hardware (FPGAs). This article reviews the language design from the outset, and highlights some of the earliest design decisions. We also describe how these decisions were revised recently to accommodate important requirements that arise in networking and cryptography.

1998 ACM Subject Classification D.3.3 Language Constructs and Features, B.5.2 Design Aids

Keywords and phrases Heterogeneous Systems, FPGA, High Level Synthesis, Dataflow, Functional Programming, Streaming, Java

Digital Object Identifier 10.4230/LIPIcs.SNAPL.2015.32

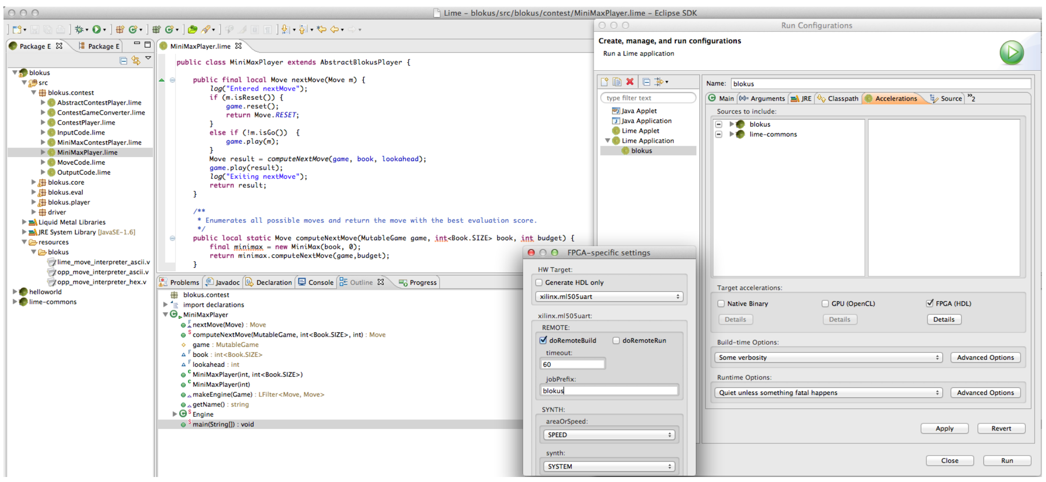
1 Introduction

Over the coming decade, the mix of computational elements that comprise a "computer" will grow increasingly heterogeneous. Already many systems exploit both general CPUs and graphics processors (GPUs) for general purpose computing. The ubiquity of GPUs, and their efficiency for certain computational problems, make them attractive hardware accelerators. Other forms of hardware accelerators include field programmable gate arrays (FPGAs) and fixed-function accelerators for cryptography, XML parsing, regular expression matching, and physics engines.

Although programming technologies exist for CPUs, GPUs, FPGAs, and various accelerators in isolation, current programming methods suffer from at least three major deficiencies. First, disparate architectures must be programmed with disparate languages or dialects, thus challenging a single programmer or programming team to work equally well on all aspects of a project. Second, current solutions provide relatively little attention to *co-execution*, the problem of orchestrating distinct computations on different architectures that must work seamlessly together. Third, current systems force an early static decision as to what will execute where, a decision that is costly to revisit as a project evolves. This is exacerbated by the fact that some of the accelerators, especially FPGAs, are notoriously difficult to program well and place a heavy engineering burden on programmers.

2 One Language, or Three

We designed Lime as a programming language for systems which connect a *host* general purpose processor (CPU) to a specialized *device* such as a GPU or FPGA. In general, the system may encompass more than one CPU, GPU or FPGA.


Lime – blokus/src/blokus/contest/MiniMaxPlayer.lime – Eclipse SDK

```
blokus
  src
    blokus.contest
      AbstractContestPlayer.lime
      ContestGameConverter.lime
      ContestPlayer.lime
      InputCode.lime
      MiniMaxContestPlayer.lime
      MiniMaxPlayer.lime
      MoveCode.lime
      OutputCode.lime
    blokus.core
    blokus.eval
    blokus.player
    driver
  Liquid Metal Libraries
  JRE System Library [JavaSE-1.6]
  resources
    blokus
      lime_move_interpreter_ascii.v
      opp_move_interpreter_ascii.v
      opp_move_interpreter_hex.v
helloworld
lime-commons
```

MiniMaxPlayer.lime

```
public class MiniMaxPlayer extends AbstractBlokusPlayer {

    public final local Move nextMove(Move m) {
        log("Entered nextMove");
        if (m.isReset()) {
            game.reset();
            return Move.RESET;
        }
        else if (!m.isGo())  {
            game.play(m);
        }
        Move result = computeNextMove(game, book, lookahead);
        game.play(result);
        log("Exiting nextMove");
        return result;
    }

    /**
     * Enumerates all possible moves and return the move with the best evaluation score.
     */
    public local static Move computeNextMove(MutableGame game, int<Book.SIZE> book, int budget) {
        final minimax = new MiniMax(book, 0);
        return minimax.computeNextMove(game,budget);
    }
```

Outline:

```
blokus.contest
import declarations
MiniMaxPlayer
  nextMove(Move) : Move
  computeNextMove(MutableGame, int<Book.SIZE>, int) : Move
  game : MutableGame
  book : int<Book.SIZE>
  lookahead : int
  MiniMaxPlayer(int, int<Book.SIZE>)
  MiniMaxPlayer(int)
  makeEngine(Game) : LFilter<Move, Move>
  getName() : string
  Engine
  main(String[]) : void
```

FPGA-specific settings

HW Target: Generate HDL only: unchecked; xilinx.ml505uart
xilinx.ml505uart: REMOTE: doRemoteBuild ✓; doRemoteRun: unchecked; timeout: 60; jobPrefix: blokus
SYNTH: areaOrSpeed: SPEED; synth: SYSTEM

Run Configurations

Create, manage, and run configurations
Run a Lime application

Java Applet, Java Application, Lime Applet, Lime Application: blokus

Name: blokus
Accelerations tab. Sources to include: blokus, lime-commons
Target accelerations: Native Binary: unchecked; GPU (OpenCL): unchecked; FPGA (HDL) ✓
Build-time Options: Some verbosity
Runtime Options: Quiet unless something fatal happens

Figure 1 The Lime IDE is an Eclipse plugin available at `http://lime.mybluemix.net`.

Consider a program as a collection of components, some of which run on the CPU, and some which run on the device. Clearly we need to describe the host program, the device program, and how to coordinate these two components and their communication. Lime allows a developer to program all three aspects using a single language and semantic rules which are neither host or device specific. This is convenient for the developer, provided the language implementation can deliver reasonable performance compared to using a plurality of languages.

Central to the Lime programming language lies the notion of dataflow graphs, where nodes encapsulate computation and edges describe data communication between nodes. Lime provides features for constructing planar and irreducible dataflow graphs which we call *task graphs*.

A task graph may execute entirely on the host or entirely on the device. Alternatively, the graph may be partitioned into subgraphs, some of which run on the host and some on the device. Thus the nodes in the graph can run on the parts of the system they are best suited for. Moreover, it may be desirable to migrate nodes between processors, in response to run time characteristics.

Lime empowers a skilled programmer to develop an application, whose parts are intended to run on different processors in the system, using a single language and semantic domain. Instead of writing code in multiple languages and using different tools to compile and assemble the source for each processor in the system, Lime makes it possible to program using a single language, a single compilation toolchain, and a single virtual runtime to execute the application on the heterogeneous system (Figure 1).

We did not always think of Lime as consisting of three languages – the host and device languages, and the task graph language – but this has proven increasingly helpful as we think about applying the salient features of Lime to other languages.

3 Salient Features

We highlight some of the Lime language features using the example code in Figure 2. The Figure shows a valid Lime program that is also a valid Java program. The class `HelloWorld` defines a method `getChar` to read a character from standard input, and a method `toLowerCase` to convert a character to its lowercase counterpart. The main method

```
 1 public class HelloWorld {
 2    public static void main(String[] args) {
 3       while (true) {
 4          System.out.print(
 5               toLowerCase(
 6                   getChar()));
 7       }
 8    }
 9    static char getChar() {
10       try {
11          return (char) System.in.read();
12       } catch (Exception e) {
13          return ’\0’;
14       }
15    }
16    static char toLowerCase(char c) {
17       return (’A’ <= c && c <= ’Z’) ?
18              (char) (c + ’a’ - ’A’) : c;
19    }
20 }
```

Figure 2 Lime example.

```
   public static void main(String[] args) {
      var tg =  task getChar
             => task toLowerCase
             => task System.out.print(char);
      var te = tg.create();
      te.start();
      te.finish();
   }
```

Figure 3 Task graph example.

composes `getChar`, `toLowerCase` and `System.out.print` to display the lowercased input characters to standard output.

The function composition on lines 4–6 describes a *pipeline*. Pipelining provides opportunities for parallelizing, where different stages of the pipeline may overlap in time to increase throughput. When composing functions in imperative or functional languages, the first stage of the pipeline is most deeply nested and appears last in an expression. Similarly, the last stage of the pipeline appears first. In contrast, task graph construction in Lime allows a more natural description of pipelines: left to right, and top to bottom.

The snippet in Figure 3 shows a Lime task graph that is equivalent to the function composition in Figure 2. Here we use the `task` and `=>` (connect) operators, and also demonstrate the use of local type inference using the `var` keyword. The task operator is applied to the same methods used in the previous example to create the tasks that make up the task graph[1]. This is akin to *lifting* in functional reactive programming [8]. In contrast to the main method in Figure 2, the pipeline is now more readily apparent. The design of the task graph language was initially influenced by the StreamIt programming language [13], although unlike StreamIt, task graph construction in Lime may describe richer graph topologies, including non-planar graphs.

A task consumes data from an input channel (if any), applies the method, and writes its results to an output channel (if any). The connect operator abstracts communication between

[1] A modifier is needed on the declaration of `toLowerCase` in order to apply the `task` operator. This is described later.

nodes in the task graph. If the entire graph runs on the host (device), then all communication is through the host (device) memory. However if say `getChar` and `System.out.print` run on the host, but `toLowerCase` runs on the device, the connect operator implies communication between the host and device memories.

The programmer communicates their intent for host and device partitioning using *relocation brackets* around individual tasks as in `([ task toLowerCase ])`, or task graphs in general [2]. We introduced this feature into the language because partitioning task graphs automatically between host and device presents too many practical and technical challenges. To understand why, note that Lime separates the graph construction (lines 2-4) from instantiation and execution (lines 5–7). This separation allows us to statically elaborate task graphs during compilation which is necessary for synthesis of hardware circuits. The hardware synthesis tools are time consuming to run, that even for this simple example it may take several minutes before a *bitfile* is produced for an FPGA. The bitfile is used to program the FPGA and realize the desired circuit. It is common to spend tens of minutes or hours in hardware synthesis for substantial programs.

In order to statically elaborate task graphs for hardware synthesis, while also using the expressive power of an imperative and object-oriented language for constructing task graphs (see [2] for examples), the Lime compiler statically checks the task graph construction for *repeatable* shape which guarantees it can extract the static structure needed for hardware synthesis. By repeatable, we mean compile-time constant but not limited to primitives [3].

The relocation brackets allow a program to constrain repeatable checks to only relevant parts of the program. Further, they facilitate gentle migration between host and device code since the programmer can move the brackets around in their code during development and testing. The feature is also useful for writing libraries because relocation brackets may be nested and composed.

The methods shown in the example may be used either in an imperative context or in a task graph context. In the former, the method is invoked explicitly by the programmer. In the latter, the task graph runtime invokes the methods repeatedly until one of a number of termination conditions is met. The two ways of using methods is convenient because the programmer can reuse existing code where necessary, and perform unit testing without invoking the task graph machinery. The dual-uses also informs the compiler of important characteristics that may be used in optimizing the implementation of a task graph. Specifically, the compiler will know the size of the input arguments and return values, and can infer an *I/O rate*. As an example, `toLowerCase` consumes one character and produces one character per invocation. Hence the compiler can use this information much in the same way that rate information is used to optimize synchronous dataflow programs [9].

More recently, we generalized the task programming model in Lime to permit the expression of tasks that directly manipulate input and output channels. In doing so, we increased the power of the language so that it is more natural to write tasks that consume or produce a variable number of values per invocation, such as those that arise in compression and networking applications. One may rewrite the `toLowerCase` method using an explicit `In` channel, an `Out` channel, or both as shown in Figure 4.

Note the use of the `local` keyword. This method modifier prohibits access to mutable global state, and hence limits side-effects of a method to those reachable through its input arguments. The local modifier is required on methods that have both input arguments and return values if they are to be lifted to task nodes. This is motivated by the desire to limit and confine side-effects for internal nodes in a task graph because they are usually migrated from the host to the device.

```
// local modifier prohibits access to mutable global data
static local char toLowerCase(char c) {
     return ('A' <= c && c <= 'Z') ?  (char) (c + 'a' - 'A') : c;
}

// using an In channel to read input characters
static local char toLowerCase(In<char> in) {
   var c = in.get();
   return toLowerCase(c);
}

// using an Out channel to return results
static local void toLowerCase(char c, Out<char> out) {
   out.put(toLowerCase(c));
}

// using both In and Out channels to read input characters and return results
static local void toLowerCase(In<char> in, Out<char> out) {
   var c = in.get();
   out.put(toLowerCase(c));
}
```

Figure 4 Task example using explicit channels.

In combination with *value* typed arguments, which are immutable types, the compiler can determine if a static method is pure in the functional programming sense. Primitive types in Lime are examples of values types. The language also provides a `value` keyword to modify class definitions and impart value semantics on a type declaration. The local modifier may apply to class constructors. This in turn permits the application of the task operator to instance methods of objects (rather than static methods), thereby creating tasks that may retain state across invocations.

All four versions of `toLowerCase` in Figure 4 are semantically identical when the task operator is applied. They vary only in the amount of rate of information they communicate to the compiler. In general, tasks that use explicit channel operations offer weaker compile-time guarantees about deadlock freedom because this style of programming inherits well-known problems attributed to Kahn Process Networks (KPN).

We further generalized the Lime task model to include non-determinism. This was motivated by functional requirements in networking applications where one may need to implement a timely merge operation. In our pursuit of a relaxed task model, we considered various relaxations of KPN to include non-determinism (e.g., [5]) as well as more general process models such as CSP [7] and Actors [1]. Most of these are broader than our needs for FPGA programming purposes. A summary of relevant literature by Lee and Parks [10] lists five ways of relaxing KPN to include non-determinism. One of these is to add the ability to test input channels for readiness. The analysis shows that this single relaxation is expressively equivalent to several others. Since any one relaxation is sufficient, we preferred the input testing relaxation because we found it to be the simplest to incorporate into our programming language.

The relaxation of the task model from its initial inception implied we could elide some of the specialized tasks in Lime in favor of library implementations. There is a trade-off however in that library functions in general may not have intrinsic guarantees and place a greater burden on the compiler to analyze their behavior for the sake of optimization.

There are many more features of the language that are not described here. We refer the interested reader to the Lime language manual [12]. The features described here highlight some of the earliest and latest design choices of the language.

4 Language Interoperability

Lime is an extension of the Java programming language and aimed for interoperability with Java from the outset, so that we may reuse existing ecosystems and libraries. Inter-language operability also made it possible to apply a gentle migration strategy to port existing Java code to Lime. In our own work, we quite often applied this strategy by starting with working Java source programs, and introducing Lime features to realize the promise of hardware synthesis.

Java was a logical choice because it is supported by mature development environments and programming tools, it executes in a managed runtime, is often optimized with a just-in-time (JIT) compiler. We thought we would get to the point of "JITing the hardware" – that is, the Lime virtual machine and JIT can exploit runtime information to further specialize the generated hardware designs, or to dynamically migrate code through a heterogeneous system in order to run code where it is best suited at a particular time.

Interoperability between Lime and Java presented a number advantages, but also some challenges particularly in the context of generics and arrays.

- The Lime generic type system does not elide all types by erasure because doing so implies the hardware backend cannot specialize the control and data paths where appropriate. In hardware design, better performance is achieved through specificity rather than genericity. Lime generic types do not attempt to be fully compatible with existing Java generics. Rather, the Lime programmer must mark particular generic types as Java-compatible where desired.
- Arrays are a important data structure in Lime, as in many programming languages. Lime revises array semantics compared to Java with two goals in mind. First, it is possible to declare arrays that are immutable (as in `int[[]]` is a read only array of integers). Second, one may establish static bounds on the sizes of arrays (as in `int[5]` or `int[[5]]` for mutable and immutable integer arrays with five elements). Array bounds guarantee that array indexing exceptions will not happen. This in turn permits simpler and more efficient hardware design. It is not feasible to achieve the expanded semantics of Lime arrays while using the identical representation to that used in Java, hence arrays (like generic types) are an area of tension between the expressivity requirements of Lime and its Java compatibility goals. This requires the programmer to make a type distinction between Java arrays and Lime arrays using designated syntax.

In retrospect, we might have used a different base language to build upon, or in an even greater departure, we could have aimed for Java interoperability but designed a minimalist language from a clean-slate. Chisel [4] for example supports hardware design as an embedded language in Scala [11]. There are of course many endeavors to raise the level of abstraction used to program hardware and commercial synthesis tools are increasingly offering viable C, C++ and OpenCL programming options for FPGAs.

5 Auto-tuning shapes

The task graph construction in Lime may use the full power of the language. That is, graph construction may use generic types and methods, recursive or iterative constructs, etc. An example is shown in Figure 5. This example introduces a few more Lime features so we will define these first. The `task` keyword appears in the method declaration to indicate this is a method that creates a task graph. This enforces a number of checks by the type system that ensure graph shapes are repeatable. Among these is a requirement that the resultant graph

```
static task <N extends int, V extends Value> IsoTask repeat(IsoTask<V,V>[[N]] filters) {
   return task split V[[N]] => task [ filters ] => task join V[[N]];
}
```

Figure 5 Example of a generic graph constructor.

is isolated – that is constructed from local methods. The type of such tasks is `IsoTask<?,?>`. We did not explicitly state this before but the type of `task toLowerCase`, for all versions of `toLowerCase` shown in Figure 4 is `IsoTask<char,char>`. The first type argument is the type of the input channel, and the second type argument is the type of the output channel.

The type parameters to `repeat` in Figure 5 are `N` which is an ordinal type (an integer), and `V` which is any value type. The former is used as a bound on the array `filters`, and also to parameterize Lime system tasks for distributing and merging values with the `split` and `join` tasks. The syntax `task [ filters ]` denotes a set of tasks that are parallel. The composition of the splitter, parallel tasks, and joiner forms a split-join which describes task parallelism akin to fork-join parallelism.

In many examples which use split-joins to parallelize computation within a task graph, it is desirable to tune the width of the split-join. In other words, one may want to experiment with different values of `N`. The Lime language design lends itself well to automating such experimentation. This is desirable because FPGAs are resource-constrained and the synthesis tools are not entirely predictable, requiring a time-consuming exploration to determine suitable split-join widths.

6 An FPGA example

Applications with a high compute-to-communication ratio generally offer opportunities for accelerators as the cost of data serialization and transfer between the host and the device can be amortized against a large, and hopefully granular, computation. One such example is homomorphic encryption [6] which allows arbitrary but bounded computation on encrypted data without leaking information about the input, output, or intermediate results. Both plaintext and ciphertext are in the form of large polynomials where the degree is in the thousands and the coefficients are hundreds of bits. At the heart of the computation are costly polynomial multiplications which dominate execution time. One acceleration strategy is to offload only these operations to an FPGA while leaving the rest of the computation and higher-level application logic on the CPU.

We implemented the polynomial multiplication using a quadratic algorithm in Lime and leveraged the support of objects to express "bigint" and polynomials as datatypes with various operations. Generics allowed the reuse of lower-level implementation types. The table below shows some results from using Lime to generate the FPGA design for a Nallatech 287 FPGA card. We report the end-to-end performance (op time), the clock frequency for the circuit in the FPGA (freq), and the resource utilization for three primary ingredients – logic cells (LUT), memory (BRAM), and DSP units which are specialized floating-point units available in FPGAs – as a percentage of total resources available in the FPGA.

	Lime
Op Time	83.3 ms
Freq	100 Mhz
LUT	11.8%
BRAM	23%
DSP	1.2%

We also implemented the algorithm in C and used a commercial high-level synthesis (HLS) tool to generate a comparable FPGA design. The performance of the Lime generated design lags approximately 2× behind that of the C generated design.

However, Lime allowed us to avoid programming against a lower level of abstraction in which object inlining was performed manually. We believe that the performance gap is due to the relative immaturity of the low-level code generation in our compiler where poor scheduling led to a relatively low frequency. LUT consumption is similar in both designs although the Lime design used far more BRAM and far fewer DSPs.

BRAM usage stems from a design with many pipelining stages and the lack of aggressive FIFO sizing optimization between tasks. In addition, the Lime design contains monitoring and features for reliability, availability and serviceability that we omitted in the C-based design.

Interestingly, despite the numerical nature of the application, the Lime design used very few DSP units, suggesting that if the BRAM issues can be ameliorated, we can horizontally scale the Lime design for greater performance. However, deeper investigation into what fraction of LUTs were used to generate mathematical units are needed to confirm.

7 Status and Concluding Remarks

The Lime development environment is available as an Eclipse plug-in [12]. We integrated support for FPGA simulation and synthesis so that the entire programming experience is accessible to software developers who have little experience with hardware design and FPGA synthesis tools. A Lime program can run on any Java Virtual Machine, so the programmer can develop and debug on a standard workstation with no specialized hardware. The compiler can translate all Lime code to JVM bytecode, and can additionally compile a *subset* of the language to Verilog for execution on an FPGA or simulator. The subset of the language that is compiled for the FPGA (or GPU) is considered the device language although there is no formal distinction between the host and device languages in Lime. The Lime compiler endeavors to make clear through diagnostics when a task that is within a repeatable graph is excluded from synthesis.

Lime encourages an incremental development path starting from Java:

1. Prototype the program in Java,
2. Add Lime constructs to express types and invariants which allow the compiler to generate Verilog for selected components,
3. Deploy the program with a JVM and a Verilog simulator for performance tuning, and
4. Synthesize a bitfile and deploy on an FPGA.

We have written more than 200K lines of Lime code to date, and developed several substantial applications drawn from many domains that include low-latency messaging, financial algorithmics, encryption, compression, codecs, and game engines. For some of these, we have benchmarked the compiler against hand-tuned FPGA implementations. One of the challenges in doing this systematically is the lack of an adequate benchmark suite with tuned software and hardware implementations to use as a baseline. Such a benchmark suite would allow us to bound performance in two ways: “from below” to measure FPGA performance compared to software implementations, and “from above” to measure generated hardware designs against manually crafted designs.

Acknowledgments. Several IBMers contributed to the project since its inception. They are Erik Altman, Joshua Auerbach, David Bacon, Ioana Baldini, Perry Cheng, Stephen

Fink, Rodric Rabbah, and Sunil Shukla. The project also benefited from contributions by Christophe Dubach (University of Edinburgh) who was a visiting scientist in 2010, and the following interns: Shan Shan Huang (Georgia Tech) in 2007, Amir Hormati (University of Michigan) in 2007 and 2008, Andre Hagiescu (National University of Singapore) in 2008, Myron King (MIT) in 2009, Alina Simion Sbirlea (Rice University) in 2011, and Charlie Curtsinger (University of Massachusetts Amherst) in 2012.

References

1 Gul Abdulnabi Agha. ACTORS: A model of concurrent computation in distributed systems. Technical Report AITR-844, Massachusetts Institute of Technology, 1985.

2 Joshua Auerbach, David F. Bacon, Perry Cheng, Stephen Fink, and Rodric Rabbah. The shape of things to run. In *ECOOP 2013 Object-Oriented Programming*, volume 7920 of *Lecture Notes in Computer Science*, pages 679–706. Springer Berlin Heidelberg, 2013.

3 Joshua Auerbach, David F. Bacon, Perry Cheng, and Rodric Rabbah. Lime: a Java-compatible and synthesizable language for heterogeneous architectures. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems, Languages, and Applications*, pages 89–108, October 2010.

4 Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC'12, pages 1216–1225, New York, NY, USA, 2012. ACM.

5 Stephen Brookes. On the Kahn principle and fair networks. Technical Report ADA356031, Carnegie Mellon University, 1998.

6 Craig Gentry and Shai Halevi. Implementing Gentry's fully-homomorphic encryption scheme. *IACR Cryptology ePrint Archive*, 2010:520, 2010.

7 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, August 1978.

8 Paul Hudak. Functional reactive programming. In *Programming Languages and Systems*, volume 1576 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1999.

9 E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. on Computers*, January 1987.

10 Edward A. Lee and Thomas M. Parks. Readings in hardware/software co-design. In *Dataflow Process Networks*, pages 59–85. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

11 Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL, Lausanne, Switzerland, 2004.

12 IBM Research. Liquid Metal Alpha Release. `http://lime.mybluemix.net`, 2014.

13 William Thies, Michal Karczmarek, and Saman P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002.