Extending Java for Heterogeneous Embedded System Description

Gary Plumbridge
Department of Computer Science
University of York, York, UK, YO10 5GH
gp@cs.york.ac.uk

Neil Audsley
Department of Computer Science
University of York, York, UK, YO10 5GH
neil@cs.york.ac.uk

***Abstract*—This paper introduces Machine Java, a framework of classes for the Java programming language that enable the description of software for systems with heterogeneous processing elements (such as CPUs, microcontrollers and function accelerators). Intended for the behavioural description of embedded systems, Machine Java encapsulates both the data and control aspects of computation into 'machine' objects that are appropriate for mapping onto architecturally diverse multiprocessors.**

System descriptions in Machine Java avoid the need for a separate programming language for each processing element, and makes explicit description of communications between processors unnecessary. Suitability for a wide variety of hardware platforms is enhanced by avoiding dependence on notions of shared memory or shared timing resources.

I. INTRODUCTION

The multiple heterogeneous components of a typical embedded platform usually require programming with several different language environments. In this paper we propose potential extensions via class libraries to Java to enable the description of whole heterogeneous embedded systems without compromising Java's sound software engineering support. The result is a Java program that can be still executed upon a conventional Java Virtual Machine (JVM) but contains enough information for translation to directly execute on hardware platforms.

The approach taken in this paper is comparable in ethos to previous approaches presented to translate single software programming languages (such as C[4], Ada[27] and SR[11]) to single hardware devices. However, the work presented in this paper is differentiated by translating a single program to a set of parallel executing devices and their interconnection pathways. These devices include CPUs, microcontrollers and FPGAs.

Good programming languages and environments already exist for many of the different classes of configurable embedded hardware, including VHDL[5] for digital circuitry description, C[15] for microprocessor programming, EDK[30] for FPGA system configuration. However, in spite of adequate tools for each system component design for a heterogeneous composite embedded system is still difficult for three primary reasons:

- Each of the heterogeneous components need different skills and knowledge to use effectively.
- Connections between the distinct components must still be explicitly defined and managed.
- Mapping of required functionality onto available hardware components must be done by hand.

In this paper we present a small set of class library extensions to the Java[12] programming language to facilitate the description and implementation of heterogeneous embedded systems in event driven style. Our primary addition to Java is a class to represent independent execution and storage, referred to as a *Machine*. Each machine embodies a computationally discrete element that can operate concurrently with every other machine in a system. A major distinction between machines and ordinary *threads* in Java is that machines may not share any memory with one another and interaction is restricted to well defined channels. The isolation between separate components represented by machines enables the possibility of mapping them effectively to both hardware devices and execution in software.

The remainder of this paper is structured as follows: Background and related work is provided in section II, the additions to Java and their implications are detailed in section III, and section IV discusses how the framework can be translated to hardware.

II. BACKGROUND AND RELATED WORK

This paper is primarily concerned with systems constructed from mixed collections of processors and reconfigurable logic, but does not exclude systems composed exclusively from either processors or logic devices. Heterogeneous system-on-a-chip architectures and systems containing CPUs or grids of FPGAs such as those used in high performance computing are of particular interest, for example the CoSMoS multi-FPGA simulation facility[28]. It is not assumed that these systems will have to be able to run a Java Virtual Machine (JVM), nor is it assumed that these hardware systems will be destined for safety critical applications.

The idea of describing a heterogeneous or multiprocessor system using a single idea has been investigated several times previously both in academia and industry. A summary of the most relevant approaches is provided in Table I. Harmonic[18] has the most similar design goals to the proposal in this paper, *Machine Java*.

Harmonic provides the ability to map a program written in a single, standard programming language, C99 onto a system composed of mixed general purpose processors and

Name	Target composition	Input language	Partitioning	Mapping	Logic generation
3L Diamond [2]	DSPs & FPGAs	C & VHDL	Manual	Manual	None
Compaan [23], [9]	FPGAs & GPPs	C & KPN	Manual	Manual	to VHDL
Hy-C [25]	FPGAs & GPPs	C	Automatic	Automatic	to VHDL
Harmonic [18]	FPGAs & GPPs	C	Automatic or manual	Automatic or manual	to Handel-C
SCF [3]	Any	Any	Manual	Manual	None
Machine Java	JVM, GPPs, μCs & FPGAs	Java	Manual	Automatic✻	to VHDL✻

TABLE I
RELATED HETEROGENEOUS CO-DESIGN APPROACHES CONTRASTED WITH THE MACHINE JAVA FRAMEWORK. ✻INDICATES FUTURE WORK.

FPGAs. Harmonic uses source-level transformation to translate input to C code that can be compiled to any GPP, and can generate Handel-C[4] code for synthesis to FPGAs. The use of standard C in Harmonic is motivated by the ability to retain compatibility with corpora of existing code. The same motivation applies to Machine Java which applies very few restrictions to the use of the Java programming language.

Both Harmonic and Hy-C[25] provide automatic partitioning of the input source code into tasks that can be mapped onto the target system's different processing elements. In Harmonic this is enabled by their use of OpenMP[20] annotations to indicate data dependancies in the code, and in Hy-C by the use of feedback of power and performance estimation from the later stages of translation to targets to inform partitioning of the source code. In contrast Machine Java does not intend to automatically partition the input system but provides constructs to the programmer to identify natural partitions in their system. The commercial approaches, Compaan [23], [9] and 3L Diamond [2] both require manual partitioning but Compaan is distinguished by it's use of Kahn Process Networks as an intermediate form for the system description.

Several of the previous approaches are capable of generating logic for FPGAs from the input system description. This enables the mapping from tasks to processing elements to be performed automatically by the toolset. Compaan is unusual in that it is capable of generating synthesisable VHDL but does not perform automatic mapping. Neither 3L Diamond nor SCF [3] are capable of automatic mapping as they require input for different target architectures to be implemented in separate programming languages. SCF is notable for addressing heterogeneous platforms by describing a common interface between components in the 'native' language of each component. SCF replaces instances of these interfaces with automatically generated code to enable the distinct components to communicate without burdening the component programmers with concrete details of the communications topology.

All of the previous approaches considered have identified that communications between unlike components is an issue and have attempted to provide abstracted communications mechanisms, but only Hy-C and Harmonic addressed the issue of having to describe the behaviour of unlike components with separate programming languages and in this way have abstracted the component-specific differences of the heterogeneous platform. However, the seemingly default choice of C as a programming language causes additional issues because of the C memory model. C depends on an assumption that memory is uniform, integrally addressed, and accessible from anywhere that code is executing. Heterogeneous platforms are likely to have memory areas that are entirely private to components or prohibitively expensive to access remotely. In the former case of private memory this does not map cleanly to the C memory model, and in the latter enables the system engineer to easily implement suboptimal memory access patterns. Additionally, two other C assumptions do not necessarily hold on some architectures such as FPGAs: That program code is in memory and that pure functions are reentrant. These assumptions do not hold when source code is synthesised to circuitry or when there is no call stack, respectively.

Java to Hardware

Perhaps the most direct Java to hardware approach was accomplished by JOP[21], an implementation of a Java virtual machine that can be synthesised onto an FPGA to provide time-predictable Java bytecode execution. Java virtual machines have been implemented in hardware many times with different design goals[19], [13], [14] indicating the plausibility of this approach. Further work has since built on JOP to enable the processor to interface with external hardware accelerators through 'hardware objects'[22] and 'hardware methods'[29]. The first abstracts hardware as objects where fields are mapped to registers in the external device, and the second interfaces with hardware by representing interaction as native methods in a similar way to the Java Native Interface (JNI)[6] in software JVMs.

Java has also been successfully applied to microcontrollers and other highly limited devices with the Java Card[24] specification aimed at the use of Java within smart cards, and the *Mote Runner*[7] virtual machine for 'mote-class' hardware which they define as microcontrollers with as little as 4KiB of ram and 32KiB of program memory. However, these approaches are not intended for multiprocessor or heterogeneous systems, and also place considerable restrictions on the Java features that can be used.

Direct synthesis of Java to digital logic for FPGAs has also been demonstrated by Sea Cucumber[26] and Galadriel[8]. While these approaches have the advantage that application-specific hardware can be described in Java itself, the supported subset for synthesis is very limited. Neither approach supports

the manipulation of objects from synthesised Java methods, exceptions are not supported, recursion is forbidden, and Java synthesised into a hardware accelerator is not allowed to call methods in software. The restricted synthesisable subset of Java ultimately renders both approaches very similar to Handel-C.

III. Machine Java

Machine Java (mJava) is a strict superset of standard Java with additional classes added to enable Java to be mapped more effectively to heterogeneous hardware. As heterogenous platforms do not always have tightly coupled components, it is considered that a programmer should be discouraged from describing the behaviour of their system in a style that requires tight coupling of processing elements either in memory or time.

An mJava program contains a set of isolated, self-contained *machines* that may only communicate via asynchronous messages. This model of computation (which is essentially an actor model of computation) motivates the following main additions to Java: A `Machine` class to represent encapsulation of data and control; Message boxes that are the communications primitive between machines; `Ports` which enable interfacing to external hardware; and Timers for providing periodic and one-time events. Intuitively, these machines are indivisible and can be mapped to any single hardware unit (such as a processor, microcontroller or FPGA). Multiple machines could be mapped to the same hardware unit or in the case of an FPGA suitable soft-cores could be instantiated to host a machine. Figure 1 provides the mapping of a simple system of machines onto a hardware device.

One of the most important aspects of the mJava framework is that it retains compatibility with standard Java. A system described with the mJava extensions is already an executable model of the target system; it can be executed in a standard JVM with approximately the same behaviour as if it were executing on the target hardware. The additions principally act as markers to assist the translation to hardware targets and during translation the mJava code will be compiled to static machine code for processors or VHDL for synthesis to an FPGA. In this paper we constrain discussion to be about the mJava framework design and semantics.

The naturally extensible nature of Java frameworks enables additional features and behaviour to be added at a later date without significantly disrupting the initial design.

A. Machine Classes

Machines provide the basis for all additions to Java and represent the logical encapsulation of memory and execution. Systems are represented in mJava as a collection of machines that may each execute concurrently and can only communicate asynchronously by sending messages to another machine's *message box*. An mJava system with exactly one Machine is equivalent to standard Java and will support all of the standard Java features and libraries when run in a JVM. See Figure 2 for an example definition of a machine. This example code also

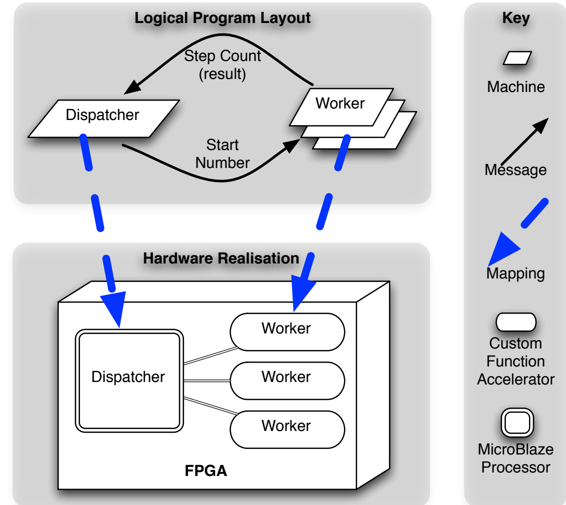

Fig. 1. The workflow of an example mJava system with a potential mapping to an FPGA based hardware platform. In this example one manager issues work over many workers.

```
1  public class Dispatcher extends Machine
2    implements Handler<Envelope<Dispatcher.
         CompletionMessage>> {
3   private final int WORKER_COUNT = 4;
4
5   @Override //Generate worker machines.
6   protected void internal() {
7    for (int i=0; i<WORKER_COUNT; i++) {
8     Worker newWorker = newMachine(Worker.class);
9     newWorker.startValueBox.sendMessage(i,
          completionBox);
10   }
11  }
12
13  final MessageBox<CompletionMessage>
       completionBox = new MessageBox <
       CompletionMessage>(this);
14
15  @Override
16  public void handle(Envelope<CompletionMessage>
       info) {...}
17  ...
```

Fig. 2. A machine definition in mJava. A class extends `mjava.core.Machine` to be recognised as a machine by the framework.

contains the definition of a message box and the prototype of the message box's handler. These are explained in detail in sections III-C and III-B respectively.

The logical layout of machines is specified in the code itself and not by some kind of auxiliary structure description file, in the example in Figure 2 it can be seen that several '`Worker`' machines are created using a static '`newMachine(`*machineType*`)`' method. To prevent objects from being 'leaked' from one machine to another, machines are only allowed to define constructors with no arguments and cannot be instantiated with the `new` keyword.

Execution and Concurrency:

The mjava.core.Machine class contains one abstract method: internal(). This method is executed immediately after the machine implementation has been initialised by the runtime and before any event handlers are executed. Other execution implications of machines include:

- Only sequential code is permitted within a machine; use of the Java Thread class is forbidden[1].
- Only one method can be executing within a machine at once, and the currently executing method cannot be interrupted by any other.
- There is no model of timing for code execution so the speed of execution cannot be used to determine timing or provide delays.
- Control flow is not allowed to move between machine objects; one machine (or library method it invoked) cannot call a method in another machine. All methods in a machine class must either be private or protected.
- Alternate models of computation can be emulated by chaining several machines to form a pipeline or multiple copies of the same machine class can be instantiated to loosely emulate thread-based concurrency.

Memory:

The mJava model of computation forbids machines to share memory[2], so object references cannot be freely passed between machines. Additionally:

- Low-level synchronisation primitives are not provided or needed in mJava as there is no mechanism for two machines to have dependancies on the same data.
- Machines are prevented from sharing objects by forcing all fields (except message boxes) in a machine class to be private or protected. Static fields are also forbidden[3] as they directly represent shared memory.

Although mJava's model is to provide interprocess communication via message passing it can be easily mapped to platforms that have shared memory rather than point-to-point communications links. It has been shown that in all cases shared memory and message passing are equivalent and interchangable[17]. While detection of memory sharing is considerably easier in Java than C as object references cannot be synthesised, the implementation of a shared memory model on platforms without shared memory hardware is expensive as updates must be propagated to every accessor of the shared data.

B. Events

Machines respond to changes in their environment by handling *events* which may be created by one of several different sources in mJava: message boxes, ports and timers. The notable aspects of events include:

- When an event source object is constructed an event handler is supplied. See Figure 2 for an example of a message box instantiation. Figure 3 shows the definition of an event handler for a message received event.
- When an event source triggers an event it is placed on the machine's event queue. These are modelled as being unbounded and ordered by arrival time, but extended implementations of the event queue can be used to change this default behaviour.
- No event arrival can preempt a the method currently executing in the machine.
- Currently executing code in a machine can chose to yield and allow queued events to be processed, program flow will continue where it stopped after the queued handlers have been executed. The yielding method can chose to execute events from a specific event source or all sources.
- Event sources can be paused to prevent the automatic execution of their handler on event arrival.

```
1  public void handle(ReturnableEnvelope<Integer,
       CompletionMessage> info) {
2   int stepCount = 0;
3   long onValue = info.payload;
4   while (onValue>1) {
5    if (onValue % 2 == 0) {
6     onValue/=2;
7    } else {
8     onValue=onValue*3+1;
9    }
10   stepCount++;
11  }
12  info.reply(new CompletionMessage(info.payload,
        stepCount));
13 }
```

Fig. 3. The message received handler in the Worker machine, 'CompletionMessage' is a simple container class of two fields, see Figure 4

C. Message Boxes

Message boxes are the only type of inter-machine communication currently supported, they enable transfer of data between machines of arbitrary size and complexity; Message boxes are strongly typed point-to-point channels between machines.

- A machine declares a message box as a public (or default) field that can be accessed by any other machine with a reference to the first. Figure 2 contains a message box declaration.
- Message boxes use Java's generic type mechanism to specify what kind of message they are capable of receiving. However, only immutable types can be communicated via message boxes. These include Java's boxed primitive types, strings, enums, machine references and classes that are annotated with @Immutable.
- The both bidirectional and unidirectional message boxes are available. Figure 3 shows the event handler for a bidirectional message box and how it has a typed return path to the sending machine.

Extended functionality is likely to be required (such as bounded box capacities, synchronous replies or flow control),

[1]Java standard libraries are allowed to use *Thread* and other forbidden classes but only on a JVM

[2]Although a runtime may chose to use shared memory as an optimisation, the code may never expect this.

[3]Again, except in Java standard libraries.

```
1 @Immutable
2 public final static class CompletionMessage {
3   final int startingValue;
4   final int stepCount;
5   public CompletionMessage(int startingValue, int
        stepCount) {
6     this.startingValue = startingValue;
7     this.stepCount = stepCount;
8   }
9 }
```

Fig. 4. A valid `@immutable` class. In the example this is used to encapsulate results back to the `Dispatcher` machine.

then these would be implemented by extending the existing `MessageBox` classes. Any functionality that is possible in a JVM can be implemented for a message box but each message box implementation with different semantics will require a new driver for each hardware target unless it can be built from more primitive boxes.

D. The Immutable Annotation:

The `Immutable` annotation can be applied to a class to indicate that it is read-only after construction. This is not a novel contribution as it has been proposed in JSR 308[10] and by Zibin et al[32] but it is not yet part of the Java specification. Immutability is useful for messages in Machine Java as it ensures a programmer cannot send data to another machine in a message that is then modifiable by both; this would violate the isolation of the machines.

Classes that are annotated `@Immutable` must obey a strict contract that dictates the class must be final and static, static fields are forbidden, all fields must be final, all non-array fields must be public and all arrays private. Array fields must be copied from a source array on construction and copied again in an accessor method. All non-array fields can only be of the following types: primitives, boxed primitives, strings, enums, machines and other `@Immutable` classes and array fields can only be of these acceptable types.

E. External Interfacing (I/O)

The ability for a machine to interface to external devices that are not abstracted by the message box mechanism is provided by `Port` classes. Ports can provide access to any hardware signals that can be represented as a collection of bits, they have the same function as signals in VHDL. `Ports` are intended to be the lowest level IO primitive possible and it is intended that frameworks of higher level IO components would be constructed (for serial ports, ethernet phys, ADCs, etc.) with ports as the core method of interaction with the hardware. An example input port is given in Figure 5.

Register classes allow the representation of a bit vector with more structure than simply representing the data as an integer. In a similar way to `Structs` in C and representation clauses in Ada, structure is provided in a register implementation by declaring fields and annotating them with their bit width and logical index within the bit vector.

```
1  public class ADC10 extends InputPort<ADC10.Value>
       {
2    public static class Value extends Register {
3      @Field(index=0, width=10, unsigned = true)
4        short value;
5    }
6
7    public ADC10(String portRef, Handler<Value>
         listener) {
8      super(portRef, listener);
9    }
10
11   public float getResistance(float upperRV) {
12      float ratio = getValue().value/1024f;
13      return ((ratio*upperRV)/(1-ratio));
14   }
15 }
```

Fig. 5. A basic representation of a 10-bit Analog-to-Digital converter in mJava. This implementation contains a convenience method to calculate the value of a resistor attached to the external device.

```
1 public static class PWM8Config extends Register {
2     @Field(index=0, width=8, unsigned=true)
3     short dutyCycle = 0;
4     @Field(index=1, width=20, unsigned=true)
5     int frequency = 0;
6 }
```

Fig. 6. A register expression of the interface to a PWM-unit in Machine Java

F. Timing

Timers in mJava are event sources that are capable of triggering events without any external interaction with the machine. Three types of timer are provided in the mJava framework:

Delays provide a constant delay between the reset of the timer and the next event arrival.

Periods provide constant delays between arrival of each event regardless of when the timer is reset as long as the timer is reset before the next event arrival is due. Period timers enable the implementation of accurate fixed-rate timers without needing to calculate the execution overheads of the event handler.

Alarms provide a single event at a fixed absolute point in the future. Resetting an alarm has no effect unless the alarm time is also changed.

IV. Translation to targets

While no specific classes of hardware have been intentionally excluded from consideration, it is envisaged that Machine Java will primarily target a mixture of JVMs, Processors, Microcontrollers and reconfigurable logic devices (FPGAs). To avoid the pitfalls of either pitching a framework at the lowest common denominator features supported by all classes of processing device or committing to implement all of Java's functionality on every target, a set of contentious features has been derived that would represent unacceptable problems during the implementation of the Machine Java translator if

Feature	JVM	Processor	Microcontroller	FPGA
RT class loading	✓	✗	✗	✗
RT Reflection	≈	✗	✗	✗
RT Polymorphism	✓	✓	✓	✗
Threads	✓	✗	✗	✗
RT shared Code	✓	✓	≈	✗
Heap allocation	✓	✓	✓	✗
Recursion	✓	✓	✓	≈
RT Machine allocation	✓	✓	✓	≈
Ports	✗	≈	✓	✓
Accurate timing	≈	≈	✓	✓

TABLE II
FEATURES LIKELY TO BE SUPPORTED BY VARIOUS CLASSES OF TARGETS, 'RT' STANDS FOR RUN-TIME. ≈ INDICATES POOR SUPPORT.

all target architectures supported the feature. A table of these features cross-referenced with the class of device can be seen in Table II.

The *processor* class of device in Table II loosely refers to modern, high-performance general purpose processors such as those with 32-bit words or more, either with or without an operating system between the Machine Java output binary and the hardware. The *microcontroller* class refers to an equally broad collection of devices that are typically harvard architecture with little memory, no external memory bus, and 8-bit or 16-bit words. Finally, the *FPGA* class refers to devices that can have VHDL synthesised for them.

V. CONCLUSION

The Machine Java framework described in this paper provides the foundations necessary to describe heterogeneous distributed systems within a single program by encouraging a programmer to solve the most challenging aspects of multiprocessor system design themselves. By providing the machine abstraction of computation a programmer partitions their own system into sections that can be much more easily distributed. The restrictions on shared memory and common notions time allow a great deal of flexibility in both compiler design and target hardware architecture.

A large amount of prior work in the fields of heterogeneous system description, compiler infrastructures, and high-level synthesis provide confidence in the feasibility of the mJava endeavour. The value of a Java based system description framework remains open to investigation, but the general notion of extending Java's 'write-once, run-anywhere' principle to hardware remains the ultimate goal.

REFERENCES

[1] "Online Encyclopedia Of Integer Sequences: A006877." [Online]. Available: http://oeis.org/A006877

[2] 3L, "Diamond User Guide," 2010. [Online]. Available: http://3l.com/technical/user-guides/3l-diamond

[3] V. Aggarwal, R. Garcia, G. Stitt, A. George, and H. Lam, "SCF: a device- and language-independent task coordination framework for reconfigurable, heterogeneous systems," in *Proceedings of the Third International Workshop on High-Performance Reconfigurable Computing Technology and Applications*, ser. HPRCTA '09. New York, NY, USA: ACM, 2009, pp. 19–28.

[4] Agility, *Handel-C Language Reference Manual*. Agility Design Solutions Inc., 2008.

[5] P. J. Ashenden, *The Designer's Guide to VHDL, Second Edition (Systems on Silicon)*. Morgan Kaufmann, 2001.

[6] C. Austin and M. Pawlan, *JNI Technology*, 2000, pp. 207–230. [Online]. Available: http://java.sun.com/docs/books/jni/download/jni.pdf

[7] A. Caracas, T. Kramp, M. Baentsch, M. Oestreicher, T. Eirich, and I. Romanov, *Mote Runner: A Multi-language Virtual Machine for Small Embedded Devices*. IEEE, June 2009.

[8] J. Cardoso and H. Neto, "Towards an automatic path from Java bytecodes to hardware through high-level synthesis," in *IEEE International Conference on Electronics, Circuits and Systems*, vol. 1. Citeseer, 1998, pp. 85–88.

[9] Compaan, "Products," 2010. [Online]. Available: http://www.compaandesign.com/index.php?option=com_content&task=view&id=3&Itemid=4

[10] M. D. Ernst, "Type Annotations (JSR 308) and the Checker Framework." [Online]. Available: http://types.cs.washington.edu/jsr308/

[11] N. Gasson and N. Audsley, "Synthesis of the SR programming language for complex FPGAs," *2009 International Conference on Field Programmable Logic and Applications*, pp. 617–621, Aug. 2009.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha, *Java(TM) Language Specification, The (3rd Edition)*. Addison-Wesley, 2005.

[13] D. Hardin, *Real-time objects on the bare metal: an efficient hardware realization of the Java Virtual Machine*. IEEE Comput. Soc, 2001.

[14] S. Ito, L. Carro, and R. Jacobi, "Making Java work for microcontroller applications," *IEEE Design & Test of Computers*, vol. 18, no. 5, pp. 100–110, 2001.

[15] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Prentice Hall, 1978.

[16] J. C. Lagarias, "The 3x+1 problem: An annotated bibliography (1963–1999)," p. 65, Sept. 2003. [Online]. Available: http://arxiv.org/abs/math.NT/0309224

[17] H. Lauer and R. Needham, "On the Duality of Operating Systems Structures," *Proc. Second International Symposium on Operating Systems*, 1978.

[18] W. Luk, J. Coutinho, T. Todman, Y. Lam, W. Osborne, K. Susanto, Q. Liu, and W. Wong, "A high-level compilation toolchain for heterogeneous systems," *2009 IEEE International SOC Conference (SOCC)*, pp. 9–18, Sept. 2009.

[19] W. Puffitsch and M. Schoeberl, "picoJava-II in an FPGA," in *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*. ACM, 2007, p. 221.

[20] M. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Science/Engineering/Math, 2003.

[21] M. Schoeberl, "JOP : A Java Optimized Processor for Embedded Real-Time Systems," *Vienna University of Technology*, no. 8625440, 2005.

[22] M. Schoeberl, C. Thalinger, S. Korsholm, and A. P. Ravn, *Hardware Objects for Java*. IEEE, May 2008.

[23] T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprette, "System design using Khan process networks: the Compaan/Laura approach," *Design, Automation and Test in Europe Conference and Exhibition, 2004. Proceedings*, pp. 340–345, 2004.

[24] Sun Microsystems, "Virtual Machine Specification: Java Card Platform, Version 3, Classic Edition," 2008.

[25] P. H. Sweany, "Hy-C Overview," 2010. [Online]. Available: http://www.cse.unt.edu/~{}sweany/research/hy-c/

[26] J. L. Tripp, P. A. Jackson, and B. L. Hutchings, *Sea Cucumber: A Synthesizing Compiler for FPGAs*, ser. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, Aug. 2002, vol. 2438, no. 2438, pp. 875–885.

[27] M. Ward and N. Audsley, "Hardware implementation of the Ravenscar Ada tasking profile," in *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*. New York, New York, USA: ACM, 2002, p. 68.

[28] J. Whitham, "The Cosmos multi-FPGA simulation facility," *CoSMoS Workshop*, 2008. [Online]. Available: http://www.jwhitham.org.uk/pubs/cosmos08.pdf

[29] J. Whitham, N. Audsley, and M. Schoeberl, "Using hardware methods to improve time-predictable performance in real-time Java systems," *Time and Embedded Systems*, 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1620405.1620424

[30] Xilinx, "Platform Studio and EDK Documentation." [Online]. Available: http://www.xilinx.com/ise/embedded/edk_docs.htm

[31] ——, "MicroBlaze Processor Reference Guide," 2008. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

[32] Y. Zibin, A. Potanin, M. Ali, S. Artzi, A. Kieun, and M. D. Ernst, "Object and reference immutability using java generics," *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 75–84, 2007.