Translating Java for Resource Constrained Embedded Systems

Gary Plumbridge
Department of Computer Science
University of York
gary.plumbridge@york.ac.uk

Dr. Neil Audsley
Department of Computer Science
University of York
neil.audsley@york.ac.uk

***Abstract*—This paper discusses a strategy for translating the Java programming language to a form that is suitable for execution on resource limited embedded systems such as softcore processors in FPGAs, Network-on-Chip nodes and microcontrollers. The translation strategy prioritises the minimisation of runtime memory usage, generated code size, and suitability for a wide range of small architectures over other desirable goals such as execution speed and strict adherence to the Java standard.**

The translation procedure, or *Concrete Hardware Implementation* of a software application first converts the application's compiled Java class files to a self-contained intermediate representation conducive to optimisation and refactoring. The intermediate format is then serialised into a programming language compilable to the target architecture. This paper presents techniques for analysing whole Java applications, translating Java methods and building a stand-alone translated application with the same functional behaviour as the original Java. An example C-code generator is described and evaluated against similar previous approaches. An existing benchmark application, JavaBenchEmbedded, is demonstrated to require less than 30KiB of program code and 16KiB of runtime heap memory when executing on a Xilinx MicroBlaze Processor.

I. Introduction

Small embedded processors such as those found in highly parallel Network-on-Chip (NoC) designs or as softcore processors in FPGAs typically only have tens of kibibytes of local memory. The restricted code and data sizes of these platforms has traditionally motivated the use of C and assembly because of the ability of their compilers and assemblers to produce compact binaries and to provide direct, low-level interaction with the hardware. In addition to restricted memory these processors may have low computational throughput and reduced functionality. In this paper a technique is presented for translating Java[8] applications so that Java can be used for programming such highly resource constrained custom systems.

Java has many advantages as a programming language for embedded systems including: strong typing, object orientation, exceptions and managed memory. Because of these features application development in Java is more rapid and more robust than using C.

Standard Java is compiled to an extremely compact and high-level bytecode but this requires a Java Virtual Machine[14] (JVM) and a large set of libraries[1], together often called a Java Runtime Environment (JRE), for execution. However, the size of the standard Java libraries is prohibitive in the context of small processors and microcontrollers. In addition to high memory costs, standard Java also has dependencies on services usually provided by an operating system such as console IO, concurrency, networking, and graphics. These services may not be provided or appropriate in an embedded processing context.

The general strategy for implementation of Java applications on embedded systems is the Ahead-of-Time compilation and optimisation of Java bytecode to compilable C. The code generated by the approach in this paper does not depend on any external libraries or the presence of an operating system. To achieve this goal three main tactics are applied to reduce the overhead of the Java environment. These are:

- Only including methods, objects and Java functionality that is used by the application in the self-contained output.
- Restricting the use of Java language features that present an obstacle to compact code generation.
- Abstraction of fundamental target system characteristics so that a code-generator can be reused for different processor architectures.

As the work presented here allows the complete software for a small embedded systems to be written in standard Java, a new advantage becomes available to systems developers: the application software will also run in a standard JRE without modification. This enables considerably faster system development and debugging of application code.

The remainder of the paper is structured as follows: Background and related works are described in section I, the *Concrete Hardware Implementation* (Chi) strategy and interpretation of Java programs is described in section III-B, the generation of C code applications from Chi intermediate form is detailed in section IV, a discussion of performance is provided in section V and finally ongoing and future work is described alongside the conclusions in section VI.

II. Background and Related Work

The Java programming language[8] has proven an attractive choice for the development of both desktop and server applications and this popularity is also reflected in the quantity of related work. A summary of relevant and interesting prior

[1] with Java 1.6 on MacOS 10.6.7 the *classes.jar* of the runtime is 30MiB.

Name	Purpose	Retargettable?	Standard Java?	Self-contained output?	Output Format
Toba [17]	Java to C conversion	Solaris & Linux	Yes	No	C
Varma and Bhattacharyya, 2004[23]	Java to C for embedded systems, included in PtolemyII[4]	Solaris, Cygwin & DSP	Yes	Yes, except native methods & GC library.	C
GCJ [6]	Compilation of Java to native code	Yes	Yes	Requires libgcj & operating system.	Native code
Nilsson, 2004 [15]	Compilation of Java for hard realtime systems	Linux & ATmega128[1]	Yes	Requires a real-time OS.	C
JEPES [21]	Compilation of Java for extremely restricted systems	Yes	No	Yes	Native Code
Liquid Metal (Lime) [10]	Compilation to embedded JVMs and FPGA logic (co-synthesis)	No	No (Immutable value-types required)	No	Java `.class`es and Verilog
Jop [19]	Java softcore processor	n/a	Simplified real-time specification	n/a	n/a
Chi	Java to specialised, self-contained C	Linux, MacOS & Xilinx MicroBlaze[24]	Yes	Yes	C

TABLE I
RELATED APPROACHES FOR IMPLEMENTATION OF JAVA IN EMBEDDED SYSTEMS.

approaches is provided in table I. The design goals of the work by Varma and Bhattacharyya[23] is most similar to that of the work presented in this paper. They detail a Java-to-C converter to enable the an application written in Java to be compiled 'though' a C compiler to native code. The use of C[13] as an intermediate language is a natural choice; practically every processor, microcontroller and DSP will have a functional C compiler. In addition to being widely supported, C compilers such as the popular GCC[5] have sophisticated optimisations built-in to minimise execution time and binary sizes without the need to build these optimisations into the Java translator.

Java has previously been applied successfully to environments with as little as 512bytes of RAM and 4KiB of program code through the use of the Java Card[22] specification, or the JEPES[21] platform. However, both the Java Card specification and the JEPES platform make significant changes to the Java environment in to enable execution on such limited hardware. In particular both of these approaches change the available standard libraries and reduce the set of available primitive types available to the programmer: Java Card simply forbids several primitives including char, float and double. JEPES makes the existence of floating point types dependent on the target platform and reduces the widths of other primitive types too.

In contrast to the use of a specifically cut-down Java specification, Varma and Bhattacharyya[23] enable the use of the full Java libraries by extensively pruning unused code from application and libraries under translation such that only used code is present in the resulting C code. Toba[17] and GCJ[6] which are not intended for use in embedded systems do not perform such code pruning. Toba, which is no longer maintained, used a strategy of elaborating the full class libraries into C. Translating a whole Java library greatly simplifies compilation and saves time as this only need be done each time the Java libraries are changed. The compiled Java library can then be linked against the user's own code without any special consideration. This is also the approach used by GCJ which requires that the user's code be linked against a large 'libgcj.so'. As of GCJ 4.4 this shared object is up to 34MiB putting it well out of reach of the capabilities of small embedded systems.

In another similar work, Nilsson[15] presents a Java to C compiler with an emphasis on support for hard real-time applications. Importantly, their work incorporates the implementation and evaluation of real-time garbage collectors suitable for including on a sophisticated, the Atmel ATmega128[1], although they report disappointing performance while the garbage collector is enabled. Nilsson also confirm the observation that the virtual dispatch of Java methods can have a significant impact in runtime performance (they observe a 43% increase in execution time in one experiment when dynamically dispatched methods are used compared the same experiment with statically dispatched methods). While Schultz et al.[21] in the JEPES work also recognise that dynamically dispatched methods come at a significant runtime price, this is dismissed as an unavoidable cost.

In spite of the aggressive code pruning employed by Varma and Bhattacharyya[23], the costs associated with the dispatch of virtual methods is not addressed and the runtime in-memory class descriptor structures are verbose (including class names, references to superclass structures, the size of instances, and if the class is an array or not) considering that reflection is not supported by their compilation process.

Huang et al. [10] introduce the *Liquid Metal* framework for the generation of JVM bytecode and synthesisable Verilog from a Java-like input language called *Lime*. The Liquid Metal procedure is able to automatically generate the interface between the synthesised hardware functional units and the software running on a JVM. While not explicitly intended for embedded systems this is a promising approach for implementing Java applications in systems with spare FPGA resources. A potential drawback of the Lime language is that

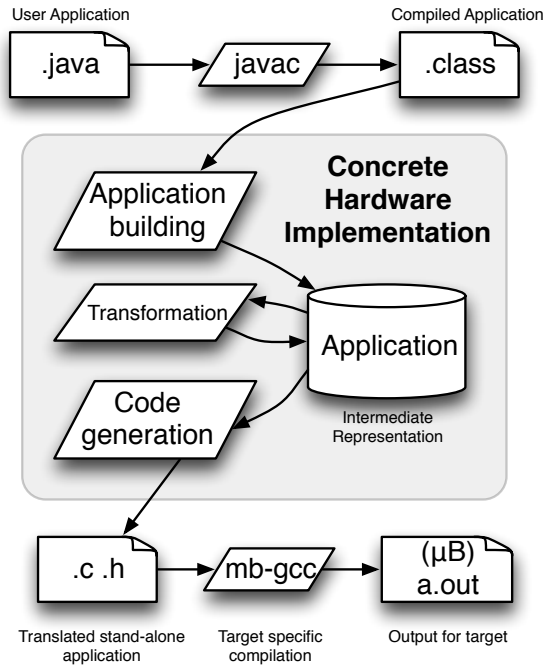

Fig. 1. An example high-level workflow of the Chi tool targetting a Xilinx MicroBlaze[24] archtecture.

the programmer is required to make use of immutable `value` types, this implies a different style of coding compared to standard Java.

Another technique for enabling Java in embedded systems is to create a processor to natively execute Java bytecodes. JOP[19] is an implementation of a Java virtual machine that can be synthesised for to FPGA to provide time-predictable Java bytecode execution. Java virtual machines have been implemented in hardware many times with different design goals[18], [9], [11] indicating the plausibility of this approach. Where an embedded system is being built around an FPGA the use of a Java machine directly is attractive as it simplifies the tool-flow from Java code to execution in the target system. However, the use of a Java softcore does not necessarily help with the problem of unsuitably large Java libraries.

III. The Concrete Hardware Implementation

The Concrete Hardware Implementation (Chi) is a strategy and related toolset geared towards enabling the full implementation of small embedded systems with Java as the input programming language. It achieves this by translating the input Java application into a target language for which a compiler already exists. All of the original behaviour of the application as it would have executed in a JVM are maintained while not requiring either a JVM or a Java standard library at runtime. Chi is able to create small, fast program code by performing operations at translation-time that would ordinarily be deferred to runtime: The interpretation of Java bytecode is performed Ahead-of-Time (AoT) during translation as opposed to Just-in-Time (JIT) compilation at runtime favoured by modern JVMs. Chi also performs AoT linking of required class files in contrast to the standard runtime demand-loading of referenced classes.

The primary intended target systems are softcore processors for FPGAs such as Xilinx's MicroBlaze[24]. Secondary targets include Xilinx's highly restricted PicoBlaze[25] processor and Plasma processing elements in the HeMPS[3] NoC architecture. In both the cases of the MicroBlaze and Plasma processors the amount of local memory available is extremely limited and of the order of 64KiB.

A. Workflow

The general workflow of Chi can be seen in Figure 1, with the five main steps being:

1) **Java Compilation** In the first stage the user compiles their Java application using the standard java compiler and libraries.
2) **Chi: Application building** The `.class` (JVM bytecode container) files generated by the Java compiler are fed to the Chi tool. The first stage of Chi is to generate an internal model of the application without including any unreachable code or unused classes. Section III-B describes this process.
3) **Chi: Transformation** After an internal 'Application' has been assembled transformations are applied to remove features unsupported by the target or code generator, perform optimisations, and prepare the application for being serialised into another programming language.
4) **Chi: Code generation** The final step in the Chi tool is to generate compilable code (in this paper, this is C). All required runtime is also generated and integrated specifically for the application at this stage. Section IV describes this process.
5) **Compilation** Finally, the target architecture specific compiler (such as GCC[5]) is invoked on the generated code.

As Chi is a translation framework for multiple different target architectures the differences between these platforms are abstracted by a 'ComputationalModel' class. The model describes to the code generator the differentiating characteristics of the target platform including native data types, endianness, alignments, available memory and peripherals. The model also describes the capabilities of the target such as support for floating point arithmetic, recursion and dynamic method dispatch.

These models are used during the application building procedure to verify that the input application is compatible with the capabilities of the target architecture. For example, if the model describes an architecture that would prefer to only statically dispatch method invocations then the application building procedure will issue an error if the application under translation requires dynamic dispatch to be behaviourally consistent with standard Java.

B. Application Building

Application building is the first stage of the Chi translation procedure where standard Java `.class` files are used to build

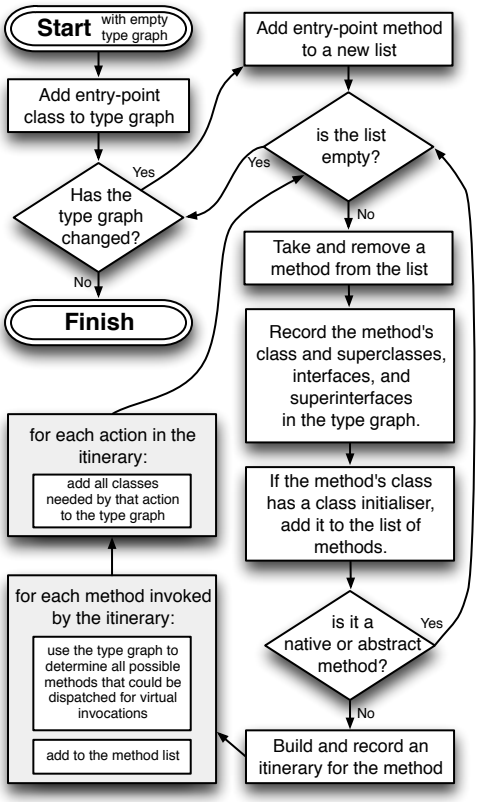

Fig. 2. A simplified illustration of the procedure used to fully elaborate only the used methods and classes in a Java application.

a complete internal representation of the user's application in memory. This internal representation includes all code that is reachable from the entry point of the user's application, including used code in the standard Java libraries if necessary.

The transformation process uses compiled Java classes to avoid the complexity associated with parsing and to take advantage of the sophisticated verification and optimisation that compliant Java compilers perform. Additionally, a considerable number of Java's language features are handled in the source compilation and are no longer present in the Java class files. Some of these features include: class nesting, generics, autoboxing, monitor handling for `synchronized` methods, and multidimentional array accesses.

The purpose of application building is to aggregate enough information about the methods and classes used such that subsequent translation steps do not have to refer back to the class files. Application building gathers the following information about a Java application:

- The entry point of the application.
- Every method callable by the application in a form called an 'Itinerary' (see section III-B). Itineraries describe the list of actions a method performs when invoked.
- Every class initialiser method (`<clinit>`) present from all classes used by the application. Class initialiser methods are used by Java to initialise default values for class static fields.
- The graph of all Java classes used by the application. This type graph considers interfaces to be superclasses of their implementations.
- The set of callable native methods. These cannot be automatically translated so subsequent code-generation has an instance specific strategy for dealing with native methods.

The application building procedure is an iterative process that can be seen in Figure 2. This procedure is essentially a variant of Bacon's Rapid Type Analysis[2] algorithm for analysis of whole applications in statically typed languages. The primary, although minor, difference is that Chi does not require an enumerable Call Hierarchy Graph (CHG) nor a Program Virtual-call Graph (PVG) as these are produced during the exploration of the input application. The application is elaborated by finding new references to classes from the currently discovered classes. This has two main advantages: Firstly that only classes actually used by the application are incorporated into the model which minimises application translation times. Secondly, the discovery procedure is able to use the running JVM's own built in class loaders. Using the JVM's class loaders to lookup classes and their bytecode allows the use of prepackaged and networked class repositories in the input application.

After all of the used classes have been discovered (including both the application's classes and those included in the JRE) they are assigned fixed, application-specific integer type identifiers. These integers are used to reduce the overhead of runtime type identification compared with retaining full information about the class hierarchy at runtime.

C. Replacements

Method calls within an application to even a seemingly innocuous library method such as `System.out.println("Hello World!")` causes a vast number of classes[2]) to become required due to transitive dependencies. This runaway in class dependencies is addressed by a framework for replacing classes and methods. Importantly, replacement classes and methods are completely transparent to the input Java application.

The computational model classes described previously contain information about which classes and methods need to have alternative implementations in order for Java to target their platform. All targets are assumed to have both `java.lang.Object` (the eventual ancestor of all classes) and `java.lang.Class` replaced. The standard implementations of `Object` and `Class` contain multiple native methods and these are either replaced with stubs to do nothing in the case of `Class` (as reflection is not implemented), or replaced with a method annotated with `@ChiNative` to indicate that the code generator must implement the functionality itself rather than use the translated the body of the method.

[2]581 classes loaded: as determined by "`java -verbose chi.tests.HelloWorld | grep Loaded | wc -l`" on Java 1.6 on MacOS 10.6.7

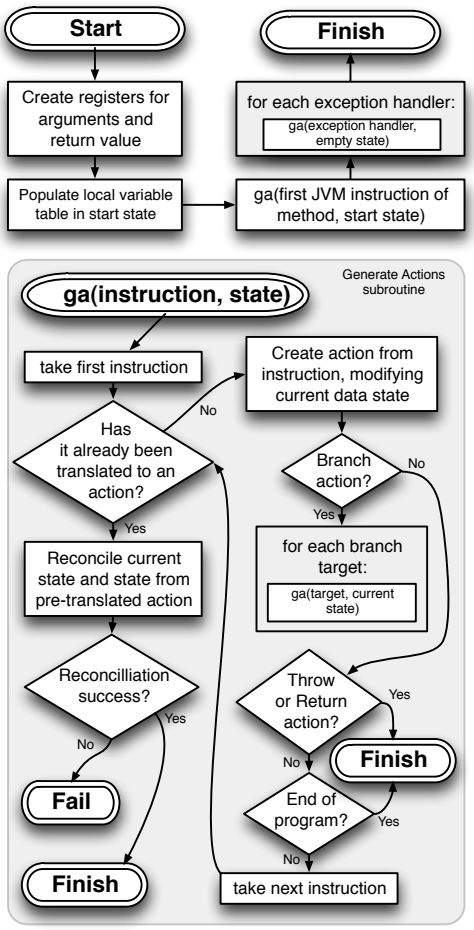

Fig. 3. The procedure used to translate Java methods in JVM bytecode to Chi 'itineraries'. Note that the greyed section of the diagram denotes a subroutine that is recursively invoked.

The method "`java.lang.Object.hashCode()`" is an example of such an annotated method.

Other classes in the set of default replacements include `String`, `System`, and `PrintStream`. The implementation of `String` in Java has an extraordinary number of dependencies rendering it far too large for small embedded systems. A considerably simpler (although non-unicode capable) replacement implementation is applied by default. In total only seven classes are in the default replacement set and these only have minor changes to remove further library dependencies (such as on system properties or security).

D. Itinerary Generation

Within the Chi tool method bodies are represented as objects called 'itineraries' which describe a sequence of actions (derived from the original Java bytecodes) upon objects and registers. The itinerary representation of methods is somewhat different to the JVM model of computation. Itineraries do not use an operand stack to pass information between actions, and there is no notion of a program counter. Where a JVM instruction would receive its operands on the operand stack, the corresponding action in an itinerary will expect its operands in specific registers (of which there are an unlimited number), and it will place any result into another specific register. Likewise `Branch` actions do not have program counter addresses of their branch targets but symbolic references to the action that would execute next depending on the situation.

The procedure used to generate Chi itineraries from a Java method is illustrated in Figure 3.

Chi uses 29 different action classes to represent the 201 valid JVM instructions. The reason for this reduction is that whole categories of JVM instructions become single actions (almost all arithmetic instructions are implemented by a `TwoOperandArithmetic` action) and all stack manipulation instructions are removed during itinerary building as there is no runtime operand stack in Chi.

All actions in an itinerary contain information such as the exception handlers that will catch an exception they might happen to throw at runtime, and the line number in the original Java source code that they were derived from.

Itinerary generation is used to gather the following information about a method:

- An ordered list of actions performed by the method.
- A list of input registers this itinerary will use to receive its arguments.
- An output register if the method is non-void to return the result.
- A set of all temporary registers used by the actions of this itinerary.

During itinerary building a notional operand stack and local variable table is maintained in a structure called the Data State. The data state keeps a mapping from the JVM operand stack and local variables to the registers which now represent the storage in the itinerary. JVM instructions that would modify either local variables or the operand stack in fact make modifications to the data state instead. When an action is generated the data state is checked to ensure that it is consistent with expectations and when the control-flow analysis leads the itinerary generation procedure back to an instruction already translated the current data state is reconciled with how the state was at the time when the instruction was first translated. Both local variable tables are compared and incompatible variables are marked as unreadable then both operand stacks are compared for consistency. If the stacks contain incompatible types in any identical positions or are different lengths then the bytecode does not meet JVM specifications and translation is aborted. This procedure is very similar to bytecode verification from the JVM specification[14].

Exception handlers appear to be unreachable code as there is no branching control flow that is able to reach them in the bytecode so they are specifically translated to actions. If the computational model has all exceptions disabled then exception handlers are indeed unreachable and they are removed as dead code.

E. Application Transformation

This paper focuses on C code generation from Java but the internal representation in the translator is not designed to

be specific to any target language. In the inevitable case that the target language has differences in operation compared to Java transformations are used to convert between JVM-like operations and the techniques used by the target language.

Action-replacement transformations are used in the cases where a target architecture does not support an operation (such as floating point arithmetic). Where these operations are used the unsupported action is replaced with a static invocation action that implements the same functionality but without using unsupported features.

Many instructions in the JVM and their corresponding actions are able to throw an exception (such as an `ArithmeticException`) without the programmer having to surround the operation in a `try{}catch{}` block. Code generation is simplified by replacing instances of these actions with a compound action that explicitly tests for the exceptional condition and then explicitly throw the exception if the condition happened.

F. Chi Runtime

The runtime architecture of a Chi application is extremely simple compared to a standard JVM. All objects and arrays are stored in memory and code generators are free to chose how to represent temporary registers. The internal arrangement of objects in memory is defined by the *ConcreteBinaryObject* (CBO) format. This format is used for all code generators and target architectures as the specific sizes of fields is dependent on the active computational model. The model provides helpers for encoding and decoding primitive types for its specific architecture. The in-memory layout of objects uses only one integer field to represent the runtime type of the object, and then each of the object's fields, in alphabetical order grouped by the class they were declared in ancestor-first order (ie: `java.lang.Object`'s fields first). If the target architecture requires data alignment then padding fields are inserted into the CBO before each unaligned field.

Arrays have one extra integer of overhead which is the number of items the array can store. This is placed after the type identifier and before the array data. It is safe for an object to omit an"isArray?" field because array manipulation operations can only be used in Java on an array typed variable. Any time there is a narrowing cast of a reference in Java, such as an `Object` reference to a variable of type `Object[]` the compiler emits a `checkcast` instruction that would throw an exception if the cast is found to be invalid. Because of these guards it is always safe to assume that a reference points to the correct type of object, even in the case of arrays.

The main operational components of the Chi runtime are the entry point and the heap allocator, both are written in Java and undergo translation along with the user's program code. The Chi entry point becomes the generated source code's entry point and performs duties such as initialising the heap allocator, initialising any device drivers that the target architecture uses, calling each class initialiser present in the application, and finally invoking the entry point specified in the user's application. This generated entry point contains the top-level exception handler responsible for catching all `Throwable` objects and printing a message to the system's console (if one exists) if an uncaught exception has propagated out of the user's application.

G. Deviations from Standard Java

Not all features of Standard Java are supported by Chi but supported features function as would be expected of a Java 1.6[8] implementation.

As a consequence of runtime types being represented by fixed integers, Java features that depend on introducing new classes into the hierarchy at runtime (such as dynamic class loading and proxy classes) are not supportable. This is because runtime type conditional behaviour is implemented as tests against constant integers which cannot be changed at runtime. To introduce a new class into an application the output code must be regenerated with the new type identification integers.

Reflection is not appropriate for deeply embedded systems due to its high memory costs and is therefore not provided. If reflection were to be supported the runtime `Class` objects must have the ability to lookup the names and offsets of all fields in an object. Reflecting on methods and constructors is also problematic as a significant portion of Chi's application minimisation depends on knowing at compile time which methods are capable of being executed. By allowing methods to be invoked reflectively all possible targets of the reflection must also be included in the output code substantially increasing code size.

Garbage collection (GC) is not yet supported but remains a matter of priority as presently applications must be very careful about allocating new objects so that heap space is not exhausted over the lifetime of a system. GC support in an embedded context has been demonstrated[15], [23] and will be implemented in due course. The Scoped and Immortal memory concepts from the Real-Time Specification for Java[7] are also a potential alternative to standard garbage collection techniques.

Support for multitasking such as threading has not been included in the current Chi procedure. Although threading is part of the Java specification it is not clear that this is the most useful model of concurrency for very limited processors, or highly distributed systems such as network on chips. Candidate multitasking approaches include the standard `java.lang.Thread` classes, `javax.realtime.RealtimeThread` from the RTSJ, or a Machine[16] abstraction of tasking.

IV. C-Code Generation

A flexible C code generator has been implemented with support for a wide variety of target architectures. Computational models have been created to enable code generation for little endian 32-bit architectures (Intel architecture) and Xilinx MicroBlaze architectures.

The general code generation strategy is to map itineraries to C functions, registers in itineraries are mapped to function-local variables of the correct type and object references are

Benchmark	JVM – JIT	JVM – Interpreted	Intel@2.4GHz	Intel – NIE	MicroBlaze@50MHz	MicroBlaze – NIE
Sieve	1,210,000 (3,470)	1,060,000 (6,270)	639,000 (49,800)	1,190,000 (138,000)	2398	3750
Kfl	6,460,000 (154,000)	220,000 (6,270)	2,710,000 (69,600)	2,860,000 (83,400)	18,886	23,355
UdpIp	1,620,000 (5,930)	95,600 (342)	1,970,000 (4,900)	2,400,000 (43,100)	6,277	9,834
Random	15,400 (262)	3,130 (11.8)	33,900 (523)	35,100 (188)	230	262
jLinpack D	425,000 (250,000)	50,900 (1,990)	418,000 (195,000)	687,000 (0)	118	128
jLinpack F	-	-	-	-	823	1,579

TABLE II
A COMPARISON OF EXECUTION PERFORMANCE OF THE BENCHMARK APPLICATIONS ACROSS A SAMPLE OF ARCHITECTURES. SECTION V DESCRIBES THE INTERPRETATION OF THESE VALUES.

mapped to the architecture's native pointer type. For each class used in the system a struct is defined according to the corresponding concrete binary object layout. Each class that has static fields is also generated a 'class instance' struct too, and for every class and array type used the integer type identifier defined in a header file.

The itinerary action structure maps very easily onto C. Each action that is an exception handler or could be the target of a branch is assigned a C label and branch actions generate optionally guarded `goto` statements.

The C generator is complicated by Java behaviours that change depending on the runtime type of an object. To save runtime data memory the code generator does not use dispatch tables to lookup which method to execute for a virtual dispatch, and the 'class instance' objects that contain the static fields of a class do not contain any information about their interface implementations or superclass. Each time a type-dependent action is encountered in an itinerary, all of the possible outcomes as determined by the previous application building procedure are enumerated. For virtual method invocations where it is determined the object on which the method is to be executed can only be of one type, the virtual dispatch is eliminated and replaced with a static dispatch instead. Where it cannot be statically determined which type an object is at translation time a switch statement is emitted that tests the runtime type field of the object against each of the possibilities determined by the type analysis.

In most cases it can be determined ahead of time what the runtime type of an object will be, but some cases such as `Object.toString()` result in huge switch statements being emitted.

All code generated depends on `stdint.h` and `math.h` for their definitions of the standard integer types and their most extreme values. The Intel architecture model also requires `stdio.h` and `time.h` for console access and determining the time, respectively. The MicroBlaze model has no extra C dependencies as it uses drivers written in Java (hidden to the user application) to provide IO and timing functionality.

The generated C application is entirely self-contained, requiring no 'libchi.a' or similar to be linked against. Each class in the original Java application (and one for each system library class used) is allocated its own C file containing method implementations. All methods, classes and types are declared across three header files. Compilation can be performed on either platform with just `gcc -std=c99 *.c`.

V. RESULTS

To evaluate the runtime performance and memory overheads of the Chi process a number of benchmark applications were used across three platforms. The benchmark tests were:

- Schoeberl's JavaBenchmarkEmbedded[20] suite including a sieve of eratosthenes (**Sieve**), a simulation of a node in a distributed motor control network (**Kfl**), and a small TCP/IP stack with two UDP clients communicating with each other via a loopback network interface (**UdpIp**). These were packaged into a single binary. Numbers in table II indicate the number of executions per second[3].
- **Random** is a test of `java.util.Random`, primitive autoboxing and ArrayList collections. Numbers in table II indicate the number of executions per second.
- **jLinpack**[12] is a port of the common Linpack linear algebra benchmark to Java. **jLinpack D** uses double precision floating point arithmetic, **jLinpack F** uses single precision. Numbers in table II indicate the number of thousands of floating point operations (kFlops) per second. There are no single precision results for the JVM or Intel architecture because the results are so large that they overflow the counter in the benchmark.

The three trial platforms used are:

1) A JVM running the input Java code, with and without Just-in-Time compilation enabled. This is on a MacOS machine with an Intel Core2 Duo@2.4GHz processor.
2) The output of the Chi tool compiled with GCC using the `-Os` option, and the binary is executed on the same machine as the JVM test. Chi was configured to use 1MiB of runtime heap. This platform is provided to enable a comparison between standard Java performance and the same application post-translation. The 'NIE' (No Instruction Exceptions) columns of table II indicates performance when runtime JVM instruction exceptions (null pointer checks, array bounds checks, checkcast, etc) are disabled.
3) A Xilinx MicroBlaze 7.30.b processor on a XC3S4000 FPGA clocked at 50Mhz.

The code sizes and runtime heap usage for these benchmark trials on the MicroBlaze architecture can be seen in table III. It should be noted that these sizes are well within the capability of most small embedded processors but are still too large for

[3]Ten trials were run in the JVM and on the Intel architecture. The figures are the mean of the trials and the figures in parenthesis in table II indicate the standard deviation of the ten trials.

Benchmark	Code Size (KiB) Instruction Exceptions Enabled	Disabled	Heap (KiB)
JavaBenchEmbedded	29.0 (1.58)	20.4 (1.58)	13.7
Random	35.7 (2.47)	20.7 (2.47)	801
Random (Standard Integer)	60.5 (2.45)	23.8 (2.45)	4.51
jLinpack	38.9 (1.87)	23.8 (1.87)	323
HelloWorld	5.64 (1.00)	4.00 (1.00)	68bytes

TABLE III
THE SIZES OF OUTPUT BINARIES AND RUNTIME HEAP USAGE.
MICROBLAZE ARCHITECTURE.

very low-end devices. The values in table III are the size of the `.text` section of the MicroBlaze binary. Values in parenthesis are the sizes of the `.data` section of the binary.

The two code sizes for the Random test in table III are for when the standard `java.lang.Integer` has been replaced with a less code-size intensive version and when the original class is being used. Although using the standard `java.lang.Integer` requires a lot of code with instruction exceptions enabled[4], the heap usage caused by the replacement class which does not cache autoboxed integer objects is unacceptably high.

It is clear from the tables that a significant performance benefit and reduction in code size is available if instruction exceptions are disabled, however without these exceptions much of Java's safety is lost. These can only be disabled if there is confidence in the correctness of the application.

Performance tests were not repeated on the MicroBlaze architecture because there is no source of non-determinism in the architecture; each execution would be clock-cycle identical to every previous one. The MicroBlaze compares favourably with results collected by Schoeberl[20] for the Jop processor at 100MHz, with Kfl and UdpIp scoring 16,591 and 6,527 respectively.

It can be seen from table II that the compiled output of Chi is similar in performance to a JVM with JIT and considerably faster than interpreted Java execution, especially when instruction exceptions are disabled. However, in three of the five comparable benchmarks JIT is faster than the compiled Java. This is likely due to the ability of the JIT to dynamically optimise the machine code dependent on the known runtime type of an object.

VI. CONCLUSION

A procedure and toolset has been presented for the execution of Java applications on highly limited hardware platforms. Code size has been effectively minimised without the use of extensive variable type analysis used in [23], providing a promising avenue for further reductions in code output code size. Execution performance was shown to be comparable between a JVM with Just-in-Time compilation and the output of Chi compiled with GCC. The minimal object representation and inline virtual method dispatch techniques do not appear to have a severely deleterious impact on runtime performance or output code size while providing substantial runtime memory savings.

The Concrete Hardware Implementation differentiates itself from similar previous approaches by an emphasis on a high-level intermediate representation useful for future investigations into both further optimisations for the reduction of resulting executable sizes and runtime memory requirements.

[4] `java.lang.Integer` uses large lookup tables to accelerate toString(), but Java class files do not have the ability to initialise arrays from a constant resulting in a huge `<clinit>` method to initialise each member individually.

REFERENCES

[1] Atmel, "ATmega128 8-Bit AVR RISC Microcontroller," 2012. [Online]. Available: http://www.atmel.com

[2] D. Bacon, "Fast and Effective Optimization of Statically Typed Object-Oriented Languages," Ph.D. dissertation, University of California, 1997.

[3] E. A. Carara, R. P. de Oliveira, N. L. V. Calazans, and F. G. Moraes, "HeMPS - a framework for NoC-based MPSoC generation," in *2009 IEEE International Symposium on Circuits and Systems*. IEEE, May 2009, pp. 1345–1348.

[4] J. Eker, J. Janneck, E. Lee, J. Ludvig, S. Neuendorffer, and S. Sachs, "Taming heterogeneity - the Ptolemy approach," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 127–144, Jan. 2003. [Online]. Available: http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1173203

[5] GNU, "GCC: The GNU Compiler Collection," 2011. [Online]. Available: http://gcc.gnu.org/

[6] ——, "The GNU Compiler for the Java Programming Language," 2011. [Online]. Available: http://gcc.gnu.org/java/

[7] J. Gosling and G. Bollella, *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

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

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

[10] S. Huang, A. Hormati, D. Bacon, and R. Rabbah, "Liquid metal: Object-oriented programming across the hardware/software boundary," *ECOOP '08 - Proceedings of the 22nd European conference on Object-Oriented Programming*, pp. 76–103, 2008.

[11] 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.

[12] M. Jeckle, "JLinpack for Java," June 2004. [Online]. Available: http://www.jeckle.de/freeStuff/jLinpack/

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

[14] T. Lindholm and F. Yellin, *Java Virtual Machine Specification*, 2nd ed. Prentice Hall, Apr. 1999.

[15] A. Nilsson, "Compiling java for real-time systems," Licentiate Thesis, Lund University, 2004.

[16] G. Plumbridge and N. Audsley, "Extending Java for heterogeneous embedded system description," in *6th International Workshop on Reconfigurable Communication-Centric Systems-on-Chip (ReCoSoC)*. IEEE, June 2011, pp. 1–6.

[17] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson, "Toba: java for applications a way ahead of time (WAT) compiler," in *Proceedings of the 3rd conference on USENIX Conference on Object-Oriented Technologies (COOTS) - Volume 3*. Berkeley, CA, USA: USENIX Association, 1997, p. 3.

[18] 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.

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

[20] ——, "JOP Performance," 2006. [Online]. Available: http://www.jopdesign.com/perf.jsp

[21] U. P. Schultz, K. Burgaard, F. G. Christensen, and J. r. L. Knudsen, "Compiling java for low-end embedded systems," *LCTES '03 Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, vol. 38, no. 7, p. 42, July 2003.

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

[23] A. Varma and S. S. Bhattacharyya, "Java-through-C Compilation: An Enabling Technology for Java in Embedded Systems," *DATE '04 Proceedings of the conference on Design, automation and test in Europe - Volume 3*, p. 30161, Feb. 2004.

[24] Xilinx, "MicroBlaze Processor Reference Guide," Tech. Rep., 2008. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/mb_ref_guide.pdf

[25] ——, *PicoBlaze 8-bit Embedded Microcontroller User Guide*, 2011. [Online]. Available: http://www.xilinx.com