Turning Java into Hardware: Caffinated Compiler Construction

C. Scott Ananian

March 24, 2002
Revision: 1.12

1 Introduction

This paper explores the design of a compiler for the Java programming language. Unlike many compilers, the target is hardware, not bytecodes or machine instructions. Java's simplicity, object-orientation, and strong typing make it well suited to class-based hardware translation. It is also possible to leverage Java's thread interfaces to model coarse-grain parallelism in hardware. The goal is the efficient generation of hardware from a well-known general-purpose programming language. The use of one specification language for both the hardware and software components of a system could also aid hardware-software codesign; this may be explored in future work.

Java, as a general-purpose portable programming language, presents obstacles to its use as a hardware specification language. The lack of integer types with parameterized bit-widths is a limitation that compiler technology can overcome. It is not certain that Java's inability to specify real-time constraints on code or describe hardware interfaces at the wire and timing level can be similarly smoothed over. This is a topic of research.

2 Java/Hardware Semantics

The hardware semantics we give Java are tied closely to its object model. As a general rule, every class instance corresponds to a hardware block, with the method call graph of the program dictating the wiring between blocks. Fully dynamic allocation is not supported at this point, as it would involve dynamically reconfigurable logic. Only static allocations can be synthesized into hardware; however, it is possible through intelligent analysis to convert most dynamic allocations into static ones. This analysis is similar to global allocation algorithms used for DSP targets [**?**] and is described further in 3.1.

2.1 Object semantics

Methods form the interfaces to the hardware generated for class instances. The mapping is one-to-one: we associate a bus with every formal parameter to the method, and another with the return value.[1] Our static allocation analysis transparently handles single-use return-value objects to allow methods to return multiple values. Methods are treated as if they are inlined in the code; that is, their logic is typi-

[1] Our IR actually represents calls as returning an optional exception object in addition to the usual return value; thus *two* buses are actually associated with the return.

cally replicated at every call.[2] The exceptions are `synchronized` methods, where the synchronization primitives arbitrate method-logic sharing. In cases where static analysis does not permit an exact determination of the object pointed to by a variable, multiplexers are inserted to perform method dispatches dynamically.

Fields in a Java class are also translated as wiring directives, utilizing renaming as necessary. Parallel field accesses should be protected in Java by monitors; the monitor synchronization process in Java is leveraged to provide access control to field values.

Java threads are detected at compile time in the static allocation phase and compiled as parallel hardware. This is a natural extension of the object semantics, as `java.lang.Thread` is a proper object in the Java environment.

Exception objects are supported, but they have no special meaning other than as a special type of method return value. They are statically allocated like all other objects.

Up to this point we have left unspecified issues of timing.

2.2 Timing semantics

The efficient implementation of Java in hardware entails a tight-rope walk between the usual sequential statement-execution semantics and a radically synchronous reading similar to the languages Esterel [4], Lucid [**?**], and SIGNAL [1]. One option is to create an entirely asynchronous design [9] from a dataflow graph of the Java program that respects the original program's sequential order exactly. Such a design has favorable power consumption and execution time characteristics, but suffers from high circuit complexity [5, 14]. Most silicon compilers instead generate synchronous circuits from their input programs, allowing us great flexibility in the assignment of clocked states to an compiled object. It seems most reasonable to use a synchrony model a bit looser than the *perfect synchrony hypothesis* used in Esterel, which Berry in [3] credits to [2]. Instead of "instantaneous broadcast" we prefer a "time plus delta" approach similar to that used by VHDL [**?**] and formalized in [11] citing [10]. The program should behave according to the standard sequential semantics, but externally-visible events will be synchronized to a clock. Conceptually, all statements take a finite but infinitesimal amount of time, excepting at points where we wait for the next clock tick.

The external event clock synchronization should follow a clear timing model so that a programmer can easily understand and specify the clock-cycle-level behavior of a Java specification. We use a variant of a scheme first proposed in [12]: back edges in a control flow graph correspond to cycle boundaries. That is, any loop back to prior code will take one cycle. Looping constructs are the only place where these back edges will be found, and they will take one clock cycle for every loop iteration.[3]

Note that only backwards control-flow-graph edges cause cycle boundaries. The first loop iteration is executed in the same clock cycle as the code preceding it, and the last loop iteration occurs in the same clock as code following it.[4] Zero- or one-iteration loops do not create clock cycle boundaries.

Registers are inserted on any wires that cross clock cycle boundaries. Although typically all object fields will become registers (in order to save state across cycle boundaries), certain short-lived fields will not be registered. The short-lifetime objects used to re-

[2] The replicated code blocks should be specially tagged as potential targets for later optimization, however. See also section 3.2.

[3] An explicit "pause for tick" statement can be synthesized from a 2-iteration loop, although this will probably be wrapped in a library method to hide the implementation.

[4] This behavior can be changed by using the pause-for-tick statement, of course.

turn multiple values from a function are a good example of fields unlikely to be synthesized as registers.

2.3 Java language coverage

Our goal is to be able to synthesize any Java program without any restrictions. However, the initial implementation will probably not support some language features.

The first language restriction is on dynamic allocation, as previously discussed. The first compiler implementation will require that all objects must be statically allocatable at compile time. Future work may lift this restriction through the use of dynamically reconfigurable hardware.

Floating-point types will probably not be supported in the first compiler, as experience has shown that they are too expensive in silicon to be of much use to the hardware designer [15]. This is not an inherent limitation of the compiler; future work may add floating point support.

All other language features are supported.

2.4 External interfacing

The means by which a Java hardware specification will specify its chip-level external interfaces has not yet been determined.

3 Compiler analyses

This section will examine some of the analysis stages needed in a silicon compiler for Java.

3.1 Extended type analysis

Static type information is essential to the compilation process, in order to enable operations and methods to be synthesized correctly. More information than just type is needed, however; we actually want to compile a static list of *all allocated objects*, and associate sets of *object instances* with variables, in addition to object types. This allows us to wire a method invocation to the proper hardware representing the object instance, or add multiplexers if more than one instance could be represented. The ideas presented in [8] are instructive, but the optimization techniques presented there are not applicable: speed is not an issue—for efficient hardware we want the most precise analysis possible. The necessary analysis has much in common with the Wegman and Zadeck's Sparse Conditional Constant optimization [16], where the constants being propagated can be imagined as constant pointers to class instances. Cliff Click's work on combining optimizations is instructive [7]: the best extended type analysis is possible only in conjunction with standard constant-propagation and dead-code analysis. The algorithm for this analysis pass will probably be based on Click's optimistic analysis described in [6].

The extended type analysis pass should produce a call-graph and list of static object instances in addition to the per-variable type and instance information. If combined with traditional data flow analysis, it should identify constants and dead code, including boolean constants resulting from use of the Java `instanceof` operator.

3.2 Functional Methods

When attempting to synchronize parallel threads using the Java monitor semantics, it will be useful to distinguish between *functional* and what we shall call *non-functional* methods. Functional methods are defined to be those without side-effects, and non-functional methods are those that do have side-effects. Synchronization can be avoided for functional methods, and the relevant logic simply replicated in parallel, but non-functional methods require

access arbitration.

Computing the "functionality" of a method is a simple matter of traversing the call graph, looking for expressions with side-effects—particularly code which assigns values to instance fields.

3.3 Optimization

Most of the standard software optimization passes will also optimize the hardware generated from a Java specification; we have already mentioned constant-propagation and dead-code elimination in section 3.1. Strength reduction will prove very valuable, as multipliers are much more expensive in hardware than adders. Granlund and Montgomery describe how to recode division as multiplication in [13]; the multipliers generated can often be subsequently decomposed as adders. Certainly multiplication and division by powers of two should be optimized.

A variety of parallelizing compiler optimizations can be utilized in addition to further expose parallelism in the specification. In particular, it will be desirable to remove unnecessary control dependencies from the code, as well as unroll loops if the programmer so requests. Code motion transformations can have a large impact on the critical path length through looping constructs.

Most of these generic optimizations should precede the extended type analysis for best effect.

3.4 Bit-width analysis

It has been mentioned that one of Java's disadvantages as a hardware description language is its lack of parameterizable-width integer types. In hardware design it is often desirable to specify exactly the width of datapaths, but Java only has 8, 16, and 32-bit wide signed integers. In previous work, the author has shown that this disadvantage can be largely rectified by an intelligent bit-width analysis of the specification. The compiler can extract a minimum-required bitwidth from the constants and expressions used. This is highly desirable for an efficient hardware implementation.

3.5 Representations

The intermediate format of the compiler is described in a separate paper.

4 Worked example

5 Conclusion

References

[1] P. Amagbégnon, L. Besnard, and P. L. Guernic. Implementation of the data-flow synchronous language SIGNAL. In *Proceedings of the ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*, pages 163–173, La Jolla, California, June 1995.

[2] A. Benveniste and G. Berry. The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE: Another look at real time programming*, 79(9):1270–1282, Sept. 1991.

[3] G. Berry. Esterel on hardware. In *Mechanized Reasoning and Hardware Design*, pages 87–103. Prentice Hall, 1992.

[4] G. Berry. The Esterel v5 language primer. Available from `http://www.inria.fr/meije/esterel/esterel-eng.html`, Mar. 1998.

[5] F.-C. Cheng, S. H. Unger, M. Theobald, and W.-C. Cho. Delay-insensitive carry-lookahead adders. In *Proceedings. Tenth International Conference on VLSI Design*, pages 322–328, Hyderabad, India, Jan. 1997.

[6] C. Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, Feb. 1995.

[7] C. Click and K. D. Cooper. Combining analyses, combining optimizations. *ACM Transactions on Programming Languages and Systems*, 17(2):181–196, Mar. 1995.

[8] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL)*, pages 222–246, San Diego, California, Jan. 1998.

[9] K. D. Emerson. Asynchronous design—an interesting alternative. In *Proceedings. Tenth International Conference on VLSI Design*, pages 318–320, Hyderabad, India, Jan. 1997.

[10] J.-R. Gagné and J. Plaice. A non-standard temporal deductive database system. *Journal of Symbolic Computation*, 22(5-6):649–664, Nov.–Dec. 1996.

[11] J.-R. Gagné and J. Plaice. The non-standard semantics of Esterel. In *Advances in Computing Science — ASIAN '97. Third Asian Computing Conference. Proceedings*, pages 381–382, Kathmandu, Nepal, 1997.

[12] D. Galloway. The Transmogrifier C hardware description language and compiler for FPGAs. In *IEEE Symposium on FPGAs for Custom Computing Machines. Proceedings*, pages 136–144, Apr. 1995.

[13] T. Granlund and P. L. Montgomery. Division by invariant integers using multiplication. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, pages 61–72, Orlando, Florida, June 1994.

[14] K. Nanda, S. K. Desai, and S. K. Roy. A new methodology for the design of asynchronous digital circuits. In *Proceedings. Tenth International Conference on VLSI Design*, pages 342–347, Hyderabad, India, Jan. 1997.

[15] N. Shirazi, A. Walters, and P. Athanas. Quantitative analysis of floating point arithmetic on FPGA based Custom Computing Machines. In *IEEE Symposium on FPGAs for Custom Computing Machines. Proceedings*, pages 155–162, Napa Valley, California, Apr. 1995.

[16] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, Apr. 1991.