SANDIA REPORT

SAND2012-10192
Unlimited Release
Printed September 2012

From OO to FPGA: Fitting Round Objects into Square Hardware

Stephen Kou, Jens Palsberg, Jeffrey Brooks

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2012-10192
Unlimited Release
Printed September 2012

From OO to FPGA: Fitting Round Objects into Square Hardware

Stephen Kou and Jens Palsberg
UCLA Computer Science Department
University of California, Los Angeles
stephen@hyarros.com

Jeffrey Brooks
Decision Support Systems
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-0576
jabroo@sandia.gov

Abstract

Consumer electronics today such as cell phones often have one or more low-power FPGAs to assist with energy-intensive operations in order to reduce overall energy consumption and increase battery life. However, current techniques for programming FPGAs require people to be specially trained to do so. Ideally, software engineers can more readily take advantage of the benefits FPGAs offer by being able to program them using their existing skills, a common one being object-oriented programming. However, traditional techniques for compiling object-oriented languages are at odds with today's FPGA tools, which support neither pointers nor complex data structures. Open until now is the problem of compiling an object-oriented language to an FPGA in a way that harnesses this potential for huge energy savings. In this paper, we present a new compilation technique that feeds into an existing FPGA tool chain and produces FPGAs with up to almost an order of magnitude in energy savings compared to a low-power microprocessor while still retaining comparable performance and area usage.

The ideas and work described in this SAND report were supported by the Campus Executive initiative of the Laboratory Directed Research and Development (LDRD) program at Sandia National Laboratories.

Contents

1 Introduction **9**

2 An FPGA-oriented subset of C **13**

2.1 Memory Model 13

2.2 C Pointers 14

2.2.1 Function pointers 14

2.3 Functions and Recursion 16

3 Objects **17**

3.1 Horizontal Object Layout 18

3.2 Vertical Object Layout 19

3.3 Space Minimization 21

3.3.1 The Object Table 21

3.3.2 Table Layout and Inheritance 23

3.3.3 Structs as Pointers 24

3.4 Runtime Type Information 27

4 Arrays **29**

4.1 Array Grouping 30

4.2 Array Referencing 30

5 Methods **33**

5.1 Type Case for Method Dispatch 33

5.2 Delegates 34

6 Optimizations **35**

6.1 Method Call Optimizations 35

6.2 Bitwidth Optimization 35

6.3 Array and Object Initialization 36

7 Experimental Results **37**

7.1 Benchmarks 37

7.2 Platform 38

7.3 Measurements 40

7.3.1 Runtime Initialization 40

7.4 Microbenchmarks 41

7.5 Assessment 41

8 Conclusion **43**

References **44**

List of Figures

2.1 A set of classes written in Virgil. 15

3.1 Memory layout of `A`, `B`, and `C` in the Horizontal Object Model. 18

3.2 Visualization of in-memory layout of the uncompressed vertical object model. 20

3.3 The hybrid object layout's object table format. 22

3.4 The general structure of the pseudopointer. 24

3.5 Our class hierarchy in this class encoding scheme. 25

3.6 An illustration of the pseudopointer, pointing to an instance of `B`. 26

4.1 Array model. 29

4.2 Array reference `struct`. 31

5.1 Dispatcher function in C. 34

7.1 Benchmark code length. 38

7.2 Experimental results. 38

7.3 Runtime initialization periods. 40

Chapter 1

Introduction

Field-programmable gate arrays (FPGAs) offer a middle point between application-specific integrated circuits (ASICs) and central processing units (CPUs). ASICs have the lowest power consumption but also the lowest flexibility: they can be used for only one purpose. FPGAs, on the other hand, typically exhibit at least an order of magnitude more power consumption than ASICs [10], but they also provide greater flexibility: they can be programmed and reprogrammed. Traditionally, mobile consumer electronics such as cell phones have used ASICs to help increase battery life time by offloading the more energy or computationally-intensive operations from the CPU to the ASIC/FPGA. However, during the past decade, consumer electronics have increasingly used FPGAs to allow, for example, easy adaptation to the many cell phone standards worldwide [1]. Furthermore, the reprogrammability of the FPGA also makes it an ideal choice for hardware that needs to be upgraded or modified often.

Currently, the benefits that FPGAs offer come at a price. While CPUs are simple to program and languages made to program CPUs are generally high-level and easy to learn, ASICs and FPGAs can only be programmed by those specially trained to use the tools and languages developed specifically for designing digital circuits. FPGA designers typically use a hardware description language (HDL) such as VHDL [13] or Verilog [2] to define the behavior of the FPGA. Although recent developments have raised the level of abstraction by allowing HDL designs to be constructed from programs written in C, the barrier of entry can be reduced even more by enabling software engineers to start at an even higher level of abstraction and program FPGAs in a paradigm familiar to many – object-oriented programming. The nascent boom in FPGA use further presses the question of how this can be accomplished.

Some work has been done to approach this problem. For example, Huang, Hormati, Bacon, and Rabbah [7] have designed a new object-oriented language aimed to target both CPUs and FPGAs. A DES encryption benchmark, when written in this language, generated an FPGA design that executed fourteen times slower than the same code running on a CPU. However, their project did not consider energy usage; but we note that since energy = (power $\times$ time), longer running times decrease the energy advantage that FPGAs offer.

Schoeberl [15] presented an implementation of a Java virtual machine in which bytecode was executed by an FPGA; however, he did not compare the performance between the FPGAs and CPUs.

Let us note that compiling an object-oriented language to an FPGA is different from specifying

a hardware design in C++ using an embedded domain-specific language, as is done by Mencer, Platner, Morf, Flynn [12].

Our approach is different from previous attempts; we wish to take an existing object-oriented language that was designed without FPGAs in mind and compile programs written in this language to C programs, which can be passed through an existing tool that converts C programs to HDL designs, which can then be synthesized on an FPGA. In particular, we want to compile *bare* object-oriented programs; that is, object-oriented programs that are written in the usual syntax without any form of special annotations or pragmas to help the compiler. We want to do this in a way that realizes a large part of the energy savings that is possible on an FPGA compared to a CPU, while still attaining comparable performance and area usage.

At first, it seems straightforward to approach our goal: first compile an object-oriented language to C, and then let the tool chain from C to FPGAs take over. However, traditional techniques for compiling object-oriented languages are at odds with today's FPGA tools that don't support pointers and complex data structures. Open until now is the problem of compiling an object-oriented language to an FPGA in a way that realizes some of the huge potential for energy savings.

> **Challenge:** compile a bare object-oriented program to an FPGA with significant energy savings compared to a CPU, while still maintaining acceptable performance and space usage.

As the starting point for our investigation, we chose the Virgil programming language and the AutoPilot tool for mapping from C to FPGAs. In this paper, we present a new compilation technique that compiles unmodified, bare Virgil programs in a way that AutoPilot can successfully produce HDL designs meant for synthesis on FPGAs.

Titzer [16] designed Virgil with the purpose of programming embedded systems and device drivers within small memory. Virgil is a strongly-typed, object-oriented language akin to Java and C#. Virgil has several features that makes it an ideal language for our project. In particular, Virgil divides computation into two phases — *initialization* and *execution*; the initialization phase involves the compiler interpreting, starting with the components which contain entry point methods, the constructors in the program. Each constructor may allocate additional memory via the `new` expression, which results in an additional constructor call. Memory allocation and the use of `new` expressions to create objects and arrays are complementarily limited to constructors only. The compiler, upon completion of the initialization phase, has a view of the entire heap of the program and can then represent the generated objects and arrays in an efficient manner. Titzer, Auerbach, Bacon, and Palsberg [17] have also explored the possibility of doing this on a full-fledged Java virtual machine. The final execution of the program, whether on an FPGA or a CPU, constitutes the execution phase.

The company AutoESL, Inc. (http://www.autoesl.com) created the AutoPilot tool for converting a subset of C into various hardware description languages for synthesis onto an FPGA chip. AutoPilot is a commercialization of the experimental xPilot system developed by Cong et al. [19, 5],

and is more robust, stable, and reliable than xPilot. AutoPilot uses LLVM [11] as its frontend and then outputs a hardware design in several hardware description languages, namely VHDL, Verilog, and SystemC [3]. After further optimization for a specific FPGA brand and model, the output can be directly imported into the FPGA manufacturer's own synthesis and layout tool, which will do the final HDL compilation, synthesis, routing, and other FPGA-specific layout tasks, after which the design can be downloaded into the FPGA and executed.

AutoPilot's subset of C excludes function pointers, and places severe limitations on regular pointers, struct casting, and the contents of structs. These limitations rule out the traditional way of representing objects as virtual method tables cannot be used (as there are no function pointers), and structs cannot be used (as there is no support for casting).

Our compilation technique successfully translates bare, recursion-free Virgil programs to the subset of C that AutoPilot accepts. We build on Titzer and Palsberg's notion of vertical object layout [18], and we use the idea of using *type case* to compile virtual method dispatch without use of function pointers [4]. On top of that, we introduce two new techniques: *grouped arrays* that overcome the other limitations of AutoPilot, and the *hybrid object layout* scheme for compression of object tables. Our implementation, essentially, is:

$$\begin{aligned} \text{OO to FPGA} \quad = \quad & \text{typecase for method dispatch} + \\ & \text{grouped arrays} + \\ & \text{hybrid object layout.} \end{aligned}$$

Our approach produces HDL designs that, when executed on an FPGA, exhibit up to almost an order of magnitude in energy savings over a low-power microprocessor, and with decent performance and competitive area usage compared with HDL designs written directly in C.

The structure of the rest this paper is as follows: in the next section, we take a closer look at AutoPilot's subset of C and the motivation behind our approach. Chapters 3–5 detail how we compile objects, arrays, and methods, respectively. In Chapter 6 we discuss further optimizations, and in Chapter 7 we give our experimental results.

Chapter 2

An FPGA-oriented subset of C

AutoPilot places several limitations on the extent of C's features that are supported that dramatically change the way object references and array references are compiled, as well as requiring a completely different approach to handling virtual method dispatch. The way that we implement these three form the core of our compilation technique. Traditionally, all three areas have been solved by using scalar and function pointers. We will show that usage of pointers in AutoPilot is significantly hampered and, as a result, we cannot use them at all. Thus, the driving force behind our approach is to represent references and virtual methods without using pointers of any kind.

This section reviews the traditional methods of compiling objects, arrays, and references to the C programming language, and how the restrictions imposed by AutoPilot make these approaches infeasible.

2.1 Memory Model

The crux of the problem lies in the fundamental difference of the memory architecture of an FPGA versus that found on CPUs. The hallmark of the FPGA memory model which separates it from that of the CPU is that the memory requirements must be known beforehand; the amount of memory allotted to a design is precisely the amount of memory that the design requires; no more is given than requested. Furthermore, this memory is fragmented into many subunits, into which data is distributed and stored. This is in stark contrast to that of CPU-based computers, where memory is a single vast, allocatable, and addressable area that can be managed by the application itself; additional memory can be requested during execution.

Because of this difference in design philosophies, a program fed through AutoPilot cannot utilize dynamic memory allocation; i.e. calls to `malloc` and `free`. The reason behind this is twofold: primarily, as stated before, FPGAs themselves limit the amount of memory available to the synthesized design. ROMs and RAMs are allocated on an as-needed basis during the *design and synthesis* phase; additional memory blocks cannot be requested during runtime. Also, dynamic memory allocation and other managed memory models generally have a negative impact on performance and therefore are not ideal for hardware programming.

As a result, AutoPilot is strongly suited for programs that have statically known memory re-

quirements. Virgil is an ideal programming language to target AutoPilot because all Virgil programs have this exact property. The initialization phase explained earlier allows the Virgil compiler to know the memory footprint of the program in its entirety. We encode this memory footprint in an way that works with AutoPilot, and emit C code for the rest of the program.

2.2 C Pointers

A language construct in C that is inseparably linked to the memory model of the underlying platform is the pointer. AutoPilot performs a series of transformations on the program itself that eventually removes all pointers from the program. It relies heavily on various static analysis techniques in order to accomplish the illusion of pointers that it offers the programmer. However, dynamic pointers, or pointers that are re-assigned to point to different data during runtime, have always represented a challenging static analysis task; it is very difficult to determine to which data various pointers will point to at various points during execution if they are passed around and re-assigned. Because of this difficulty, there are several quite hefty restrictions placed on pointers by AutoPilot.

The transformation away of pointers is a necessary step taken by AutoPilot because pointers, in their traditional sense, cannot be easily implemented on FPGAs for architectural reasons. On-chip FPGA memory is not a single, large, addressable memory space like those on computers. Instead, the memory space is fragmented into many small 'blocks'. These blocks, called BRAMs in standard FPGA terminology, each have their own input/output pins, which allow for parallel reads and writes over multiple blocks. This memory architecture, while good for performance, is the reason why pointers are so hard to emulate on FPGAs.

This is compounded by the fact that each memory block has its own address space; when a pointer pointing to data residing on one block is re-assigned to data residing on another block, AutoPilot must be able to determine the correct block that holds the new data. The hardware design explicitly reflects this, as there must be a connection made in the form of a bus between the entity and the memory blocks to which it accesses. It is AutoPilot's job, then, to determine statically to which data each pointer points and route the electrical signals to the correct block when dereferenced. Pointer arithmetic, too, must also be transformed away and converted to direct data accesses as well. Therefore, pointers that are dynamically re-assigned or those whose data cannot be determined statically are not supported in AutoPilot. The end product of this operation is a program that is left without pointers at all. Any use of pointers that cannot produce this end state cannot be handled by AutoPilot. `struct`s are also limited in this fashion; they cannot contain pointers.

2.2.1 Function pointers

The standard execution model on a computer involves execution of instructions stored in the main memory. Each instruction has a corresponding address; moving between areas of code is accom-

```
class A {
  field child : A;
  field value : int;

  method foo() : void
  { }

  method bar() : void
  { }
}
```

```
class B extends A {
  field other: C;

  method bar() : void
  { }

  method arg() : void
  { }
}
```

```
class C {
  field a : int;
  field b : int;

  method f() : void
  { }
}
```

Figure 2.1. A set of classes written in Virgil.

plished using jumps to different addresses. Calling a function involves first copying arguments onto the stack then transferring execution to the address of the first instruction of the function.

AutoPilot takes a completely different approach to converting functions onto an FPGA. Each C function in the program is converted into a *design entity*, a language construct present in all HDLs that provides a level of abstraction. Like functions, which group together code and provide a black-box interface, entities also group together circuitry and logic and provide a black-box interface in the form of input and output pins. Copies of the entity, called *instances*, can be placed onto the chip, each occupying a certain amount of area. An HDL design consists of an interconnected network of entity instances.

Unlike functions on a computer, any function call rendered in this scheme requires a direct, physical connection of wires between the caller instance and the callee instance. If AutoPilot cannot determine the entity instance to which a call is made, it cannot handle that function call. This is apparent in the case of function pointers. If a AutoPilot cannot determine during compilation which function a function pointer is pointing to, it is unable to create the necessary connections that implement the call. In fact, AutoPilot at this time has no support for function pointers, even those that can be statically reasoned about.

This limitation creates substantial problems for the way the semantics of virtual methods are implemented. The traditional technique of handling virtual dispatch is the use of virtual method tables. Virtual method tables are lists of function pointers that point to the correct function to be called when a virtual method is invoked. The cell representing the virtual method is read, and the address stored within is then jumped to and executed. Because of the dependence of this approach upon the function pointer, we are forced to use a different mechanism to accomplish similar functionality and semantics.

2.3 Functions and Recursion

As stated earlier, each function is converted into design entities, of which instances are created; wires linking instances together reflect calls between different entities. This paradigm for representing functions precludes the need for stack frames to be set up in memory; here, the closest concept of a 'stack' is the area available on the FPGA chip.

FPGA tools are able to minimize the number of instances needed of each entity by allowing reuse of instances. As long as the instance is used only by one call at a time, the instance may be used over and over again. However, there are scenarios where the instance cannot be reused, such as parallel calls to the same entity. The general solution for these scenarios is to place another copy of the entity on the chip, thereby resulting in two instances of the same entity allowing for two simultaneous calls. Of course, this then leaves less remaining free area on the FPGA chip for other logic.

One significant case where this approach fails, however, is in the case of general recursion. Because the entity then needs to call itself, the existing instance is not reusable. This creates an irreconcilable case for AutoPilot: because it cannot conclude how many times the recursive call will be performed, it then assumes an infinite number of instances would be needed. However; the area on a chip is far from infinite, and thus AutoPilot rejects the input program. The design of Virgil, however, allows recursion; we do not attempt to resolve this problem in this paper. While it is possible in some cases to eliminate recursion through program transformations, it is beyond the scope of this paper and thus is relegated to future work. Other than recursion, however, functions are fully supported in AutoPilot's subset of C.

Chapter 3

Objects

We now turn to presenting our approach to compiling Virgil. Our chief concern is the representation of objects and object references. The representation method is particularly important in the case of Virgil because, like Java, programmers must use the object-oriented paradigm. *Objects* are instances of classes declared in Virgil and are allocated and instantiated in the constructors. *Object references* are pointer-like constructs that provide a level of indirection to objects; like Java, all manipulations done with objects in Virgil are through the use of references. Object references are passed, by value, to methods; object references are used in all local variables, arrays, and fields which are typed as being an object instance.

Virgil classes are relatively simple compared to their counterparts in Java, C#, or C++. Fields and methods can only be marked as public or private, while all classes can only be public. Static fields and methods are not supported, but the functionality is present via the use of singleton `components`. All classes are single-inheritance with methods being virtual by default; interfaces and abstract methods are not supported. Another important distinction between Java's and Virgil's class semantics is that not all of the classes in Virgil ultimately inherit from a common `Object` class as they do in Java. Instead, a program can consist of a set of disjoint class hierarchies, each with its own unique root class. To illustrate this, a small set of classes have been declared in Virgil syntax in Figure 2.1. We have described two distinct class families: that of `A` and `B`, and a single-member family `C`. Because the two do not ultimately inherit from a common type, they are wholly independent of each other.

The next few sections will build examples off of this sample class hierarchy. To give a quick summary of the way these classes are structured: both the fields `child` in `A` and `other` in `B` are references to other objects. Polymorphism rules allow the recursive `child` field in `A` to refer to either an instance `A` or an instance of `B`. Two methods are defined in `A`: `foo` and `bar`. `B` provides an overridden implementation for `bar`, but not `foo`; it also adds another method `arg`.

With this in mind, in the following sections we discuss and review various object layout strategies, culminating with the representation we use in our compiler that tries to minimize space while at the same time attempting to incur a minimal amount of performance overhead on an FPGA.

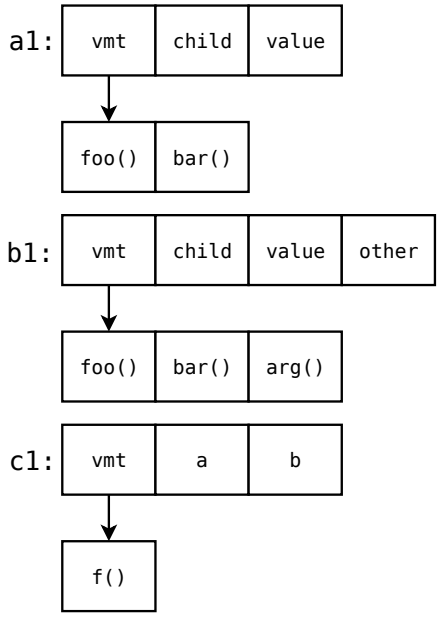

Figure 3.1. Memory layout of `A`, `B`, and `C` in the Horizontal Object Model.

3.1 Horizontal Object Layout

We term the traditional method of compiling objects to C as the ”horizontal object layout”. It is the most straightforward approach to representing objects using C language constructs, and is considered well-known; thus, we will simply provide a cursory review of this approach. The horizontal object model converts each class into a `struct`; the `struct` is composed of, in this order: (1) a pointer to a virtual method table, (2) the fields of the parent class, and (3) the fields defined in the class itself. As a result, each class' `struct` include its parent class' `struct` as a prefix.

In this scheme, references to objects are rendered as pointers which point to instances of the `struct`. Polymorphism is accomplished by exploiting the property stated earlier that the in-memory layout of a child class is compatible with that of its parent; a pointer typed as the `struct` of the child class can be casted to a pointer typed as pointing to an instance of the parent class' `struct`; all accesses to the fields and entries in the virtual method table would be compatible. A type cast is implemented in a similar fashion: a pointer typed as pointing to an instance of the parent `struct` may in reality be pointing to an instance of the child `struct`; thus, a cast can be performed in the C code on the pointer itself to cast it back to a reference of the child class.

Figure 3.1 illustrates how one instance of `A`, one instance of `B`, and one instance of `C` from our example class hierarchy would be laid out in memory using the horizontal object model. The field called `vmt` is a pointer to the virtual method table, which contains the collection of function

pointers that refer to the virtual methods. A field access is easily translated using the horizontal object model. We show a translation of a small Virgil method to C in the following code snippet. It is a simple dereference of the object pointer, coupled with a field access:

```
method f(obj : A) : int { return obj.value; }

int f(struct A* obj) { return obj->value; }
```

Although this is the most straightforward way to represent objects on a computer, it uses pointers profusely to accomplish the various traits required for inheritance and polymorphism. This object representation breaks down when AutoPilot is involved in a number of ways: function pointers like the ones used in virtual method tables are not allowed, `struct`s containing pointers are prohibited, and the casting of structs, done here to accomplish subtype polymorphism and type casting, cannot be used. It is therefore clear that we cannot use this form of object representation when targeting AutoPilot.

3.2 Vertical Object Layout

The first step forward is to first remove the necessity of the pointer in the representation of an object reference. A strategy that accomplishes this goal, and the basis for our approach, is the ”vertical” object layout [18]. We work here with the concept of the *uncompressed* vertical object model, which omits the compression scheme performed in Titzer's work.

The vertical object layout re-organizes all of the objects in a program into a large table, where the rows are the class fields, and the columns are the individual objects. There is thus one row for every field defined in the program. The virtual method table is also encoded as rows; one row for every virtual method defined. Therefore, an object in this model is a single column across all the rows. References to objects, then, do not use pointers — a reference is an integer that refers to the column index in the table where the values reside. Instead of pointers, this integer is used anywhere that an object reference is expected. A field lookup using the uncompressed vertical object is shown in the code snippet below. We show a translation of a small method into C:

```
method f(obj : A) : int { return obj.value; }

int f(int obj) { return Row_A_Value[obj]; }
```

The emitted C code is achieved from the following steps:

1. Given the field, the compiler determines the correct row to access in the object table: `Row_A_Value`.
2. The row is accessed given the object identifier: `obj`.

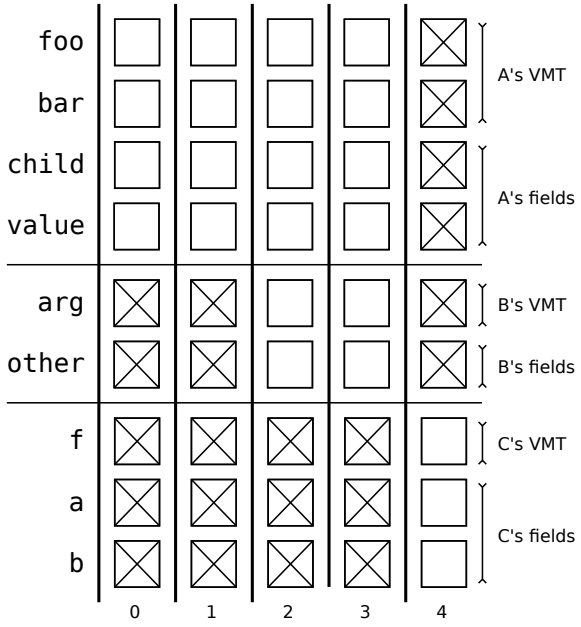

Figure 3.2. Visualization of in-memory layout of the uncompressed vertical object model.

3. The field is accessed by indexing into the field array.

Fields are no longer grouped into `structs` to reflect how they were declared in classes; instead, the vertical object model flattens the structure of the program from a set of classes to instead a set of fields. Field accesses and encapsulation are instead guaranteed by the Virgil type-checker, and such traits of the program are not reflected in the C code. Because of this property of the vertical object model, both polymorphism and casting are trivial and require no special treatment. No type information is retained in the C code, as all object references are now an `int`.

Figure 3.2 illustrates how the vertical object model would lay out 2 instances of `A`, two instances of `B`, and one instance of `C` from our example class hierarchy shown earlier. The virtual method table's entries are also rendered as rows in the object table; they are typed as arrays of function pointers.

An obvious drawback of the uncompressed vertical object model is the amount of wasted memory incurred when rendering the object table. All the crossed boxes in the figure are cells present in the table, but are unused. For example, an instance of `A` gets allocated all the memory needed for the fields of `B` and `C` as well. Therefore, as larger programs are created with more complex object hierarchies, the amount of wasted memory will grow. This design choice allows faster lookups and simpler object reference representation, at the expense of an increased memory footprint.

Nevertheless, the vertical object model solves one of the major obstacles presented in the pre-

vious section to representing objects in FPGAs and AutoPilot: it allows us to create references to objects without pointers. However, the vertical object model alone is not enough to satisfy all the limits of AutoPilot: it still employs function pointers to implement virtual dispatch, and the amount of wasted memory is not ideal for hardware, where increased area directly translates to increased costs. In the next sections, we address both of these issues by modifying the uncompressed vertical object model.

3.3 Space Minimization

The vertical object model employs a compression scheme which we do not explore in this paper. Instead, we present a compression scheme which fits well with the unique architectural properties of FPGAs.

Our aim is to compress the object table while still finding a way to represent an object reference that retains the simplicity of the vertical object model. To accomplish this, we totally restructure both the object table and the object reference. The section describes this approach.

3.3.1 The Object Table

While we retain the concept of the object table from the vertical object model, we restructure it in such a way the leaves no wasted space. Originally, every object in the object table shown in Figure 3.2 was the same size in memory — each object occupied an entire column. By compressing the object table and removing the unused cells, as shown in figure 3.3, the unused space is gone, but objects are now no longer at a single index. For example, the third column in the table, an instance of `B`, occupies the third cell in the row for `A`, but occupies the first cell in the row for `B`. Such an object table requires us to store offset information for the different rows.

One approach is to introduce another table, which maps single integers to a collection of offsets. This scheme would preserve the single-integer property of the vertical object model, while allowing us to remove the wasted space. However, this would have adverse effects on performance. A field lookup in this scheme would entail the following steps:

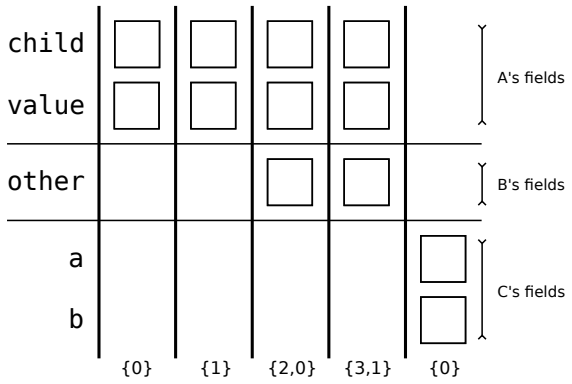

Figure 3.3. The hybrid object layout's object table format.

```
method f(obj : A) : int { return obj.value; }

int f(int obj) { return Row_A_value[Objects[obj][1]]; }
```

1. Given an index `obj`, a lookup is done into the second table to retrieve the correct offsets corresponding to the given object instance index: `Objects[obj]`.

2. Given the field being accessed, the compiler determines which row is the correct row that corresponds to the field: `Row_A_Value`.

3. The compiler determines which offset is the correct one that corresponds to the row. Here, we assume the offset at index `1` refers to the `A` row.

4. The row is then accessed, being given the correct offset retrieved from the first step. Since it is a field array, the access is simply an array access.

Two new steps are introduced: first, an additional lookup to retrieve the set of offsets; second, a determination must be made as to which offset corresponds to which row. While the latter is accomplished within the compiler, the former must occur at runtime. For traditional computer programming, such a design would be a tradeoff of performance in favor of memory. The memory minimization comes at the cost of the additional lookup for every field access.

Surprisingly, we find that this extra lookup can actually be eliminated when targeting FPGAs because of a unique optimization that can be performed. If we eliminate the second table altogether, but pass along the set of offsets directly instead of an integer to the second table, no additional lookup is needed. An object reference then would not a single integer, but instead would be the tuple of offsets that point into the object table directly. Thus, field accesses in our model consist only of steps 2-4.

To see why this is discouraged on microprocessor-based systems but a perfectly valid design choice when programming FPGAs, one must re-examine the implications of the differences in the way function calls are implemented on the FPGA by AutoPilot and the way they are traditionally done on a microprocessor. Normally, passing large data structures (larger than the register size of the platform) as arguments for function calls is discouraged. Calls such as these will both require a larger stack frame to hold the data, as well as additional copy operations to copy the data structure into into the stack frame before the function can be called. Furthermore, this is compounded by the fact that when dealing with data larger than the multiprocessor's register size, multiple loads and stores are required to move the data. Because of this, functions with complex parameters have traditionally been represented indirectly by pointers to mitigate this problem; a pointer always fits into the register and can be copied in one operation.

In the hardware world, however, no such difficulty exists in function calling. A large data structure being passed by value is just synthesized as a wider bus between the caller and the callee. Data is passed along this bus in a single parallel write operation regardless of bit width. Because of this unique property of an FPGA, widening the bitwidth of an object reference from a single integer to multiple integers incurs nearly no performance overhead. In fact, we find that for larger programs with big class hierarchies, this actually offers *increased* performance over the uncompressed model, as will be discussed in our benchmarks section.

The next subsections discuss in more detail our usage of tuples as pointers, as well as outline the other difference in our object table format versus that of the vertical object model.

3.3.2 Table Layout and Inheritance

We call our object model the *hybrid* object model because we retain the `struct` concept from the horizontal scheme, but apply it to the design philosophy of the vertical scheme. Like the horizontal object model, we take each class and convert it to a `struct`. However, instead of prefixing the struct with that of its parent, we omit the parent's layout altogether, and just include the fields immediately declared in the class itself. Thus, unlike the horizontal object model, inheritance is *not* encoded within the `struct` itself.

Each class `struct` forms a row in our object table. Because we group by `structs` instead of fields, our object table has a fewer number of rows than that of the vertical object model. Virtual method information is also omitted from the table. Objects are placed into the rows that correspond to their inheritance chain as illustrated in figure 3.3: the table layout consists of two instances of `A`, two instances of `B`, and one instance of `C`. An instance of `A` only has an entry in the row that corresponds to `A`, whereas an instance of `B`, because its parent class is `A`, consists of an entry the `A` row as well as an entry in the `B` row. Finally, an instance of `C` has only one entry in the row for `C`. Boxes represent actual elements in the row; empty spaces do not use any memory and are present to make the diagram easier to understand. Each column in a diagram represents one object instance; the text underneath shows the tuple of offsets required to accurately reference the object.

Another thing to note is that while in the horizontal object model, inheritance is apparent in

```
struct Ptr {
  char null;
  int comp1;
  int comp2;
}
```

Figure 3.4. The general structure of the pseudopointer.

the layout of the `struct` itself: the parent class' fields prefix those of the class itself, resulting in a memory layout that is compatible with that of the parent. The vertical object model, similarly, makes the class hierarchy visible in the table itself. A class instance contains fields for itself as well as its parent (and all the other classes in the hierarchy, too). For our hybrid object model, however, inheritance is shown neither in the structs nor the object table. Instead, inheritance is handled in the *object references*; the tuple contains offsets for each node in the chain from the root class to leaf class. The root class of any object will always be at the first element in the reference tuple, and child classes follow, forming the complete chain from root to leaf.

3.3.3 Structs as Pointers

These tuples of offsets which are used as references to objects we call *pseudopointers*. Though they perform the function of pointers and provide a means of indirection to objects similar to pointers, the semantics are very different.

The pseudopointer is rendered as a `struct` in C. The general structure of the pseudopointer is shown in figure 3.4. It consists of a `null` flag to indicate a reference to `null`. If nonzero, it is understood to be a null reference. Following the `null` flag are a series of integer fields `comp1` ... `compN`, which we call *component offsets*. These component offsets point to various cells in the object table. The set of cells that these component offsets point to comprise the object instance. Each component points to an element in a unique row in the table.

All objects in our program use this `struct` definition as the canonical object reference; it is used wherever an object reference is expected — local variables, fields in classes and components, elements in arrays of objects, and parameters to functions. The struct is passed by value into any functions that take object references as parameters. One example of this is shown in figure 3.5, which in particular demonstrates how the references to other classes (fields `child` and `other`) are rendered in C.

The interpretation of the component offsets is dependent on the static type of the object reference during compilation. Each class is broken down into multiple *components*; a component is a node in the class' inheritance chain. These components are then ordered from root class to leaf class.

```
struct A {
  struct Ptr child;
  int value;
};

struct B {
  struct Ptr other;
}

struct C {
  int a, b;
}
```

Figure 3.5. Our class hierarchy in this class encoding scheme.

The number of component offsets stored in the pseudopointer struct is determined by class with the greatest number of components — that is, the class with the longest path to the root class of its family. All objects which have a shorter path to the root node will use the same pseudopointer struct, but some of the component offsets will be unused.

This scheme results in an absolute ordering of components for every class: each class' ordering is prefixed by that of its parent. This property is exploited in order to accomplish polymorphism. When a reference of type `B` is assigned to a reference for `A`, it still results in a compatible pseudopointer, because the component ordering for `A` is a subset of that of `B`; all the component offsets still correspond to the correct rows.

With our sample class hierarchy, `B` has the longest chain with a length of two – the chain from `B` to the root is {`A`, `B`}. Thus, the program written using our sample hierarchy will require a pseudopointer with two component offsets, as shown in figure 3.4. All pseudopointers for references to `B` will have its two component offsets pointing to the rows for {`A`, `B`}, respectively. References to `C` will use one component out of the two {`C`, `-1`}, where `-1` is used to indicate an unused component in the pseudopointer. Figure 3.6 illustrates how the pseudopointer is interpreted for a reference to a `B`.

A field access in this scheme is translated into C as shown in the code snippet below. We convert the same mini-method that we have been using throughout this paper to show how a field is accessed:

```
method f(obj : A) : int { return obj.value; }

int f(struct Ptr obj) { return Row_A[obj.comp1].value; }
```

The generation of this code involves several steps:

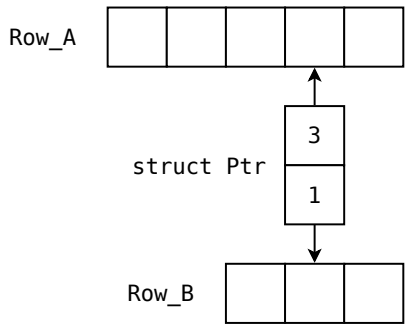

Figure 3.6. An illustration of the pseudopointer, pointing to an instance of `B`.

1. Given the field that is being accessed, the compiler determines the class that defines this field: `A`.

2. The compiler gets the position of the class within the class hierarchy. This ordering determines which offset to use in the pseudopointer: `obj.comp1`.

3. Code is emitted accessing the row that corresponds to the class using the correct offset from the pseudopointer: `Row_A`.

4. The field can then be accessed: `value`.

A component ordering for every class is created during compilation. We use this ordering also to implement polymorphism. The horizontal object model implements polymorphism through a form of structural subtyping – a child class' `struct` contains all the methods and fields, in the same order, of its parent. This allows pointers to the `structs` to be casted and still compatible. We use the class ordering itself to implement polymorphism. A child class' offset ordering is a superset of that of its parent.

As stated earlier, the compiler must create a mapping from offset position within the pseudostruct to a certain row. This mapping differs for each class family; the first offset in the pseudopointer for a reference to an instance of `B` would refer to the row for `A`; however, for a reference to an instance of `C`, the first offset refers to an element in the row for `C`. We order these components in such a way that the orderings for a subclass is always compatible with the ordering of the parent class; this way, subtype polymorphism is allowed. A reference to a `B` can be passed to method expecting a reference of type `A` as a parameter. Compatibility between parent and child class is accomplished by ensuring that the order of the parent class is a subset of that of the child. This way, when code expecting a reference to `A` received instead a reference to `B`, the offsets still correspond to the same rows. By this same design choice, casting is possible and requires no additional code.

3.4 Runtime Type Information

Augmenting this object layout model is the runtime type information field `TYPEID` that is placed into each root class' struct in order to supply Virgil's type query operator, `instanceof`, with the needed information. This field also allows us to check for incompatible type casts. The `TYPEID` field is an unique integer that is assigned during compile time to each class.

The way in which these values are ordered allow us to optimize the work needed for comparing two types for compatibility: each class knows both the minimum value of its subtree (itself) and the maximum value assigned to its children. The implementation of `instanceof` is straightforward — given an object type id, checking to see that it is greater than the id of the target class and less than the maximum value for that class will yield the correct result.

In our example, the compiler would assign a class identifier of 1 for `A`, 2 for `B`, and 3 for `C`. The `instanceof` operator querying whether an instance of `C` is compatible with `A`, then, would be rendered as follows:

```
return Row_C[obj.f0].TYPEID >= 1 &&
       Row_C[obj.f0].TYPEID <= 2;
```

Similarly, a type cast can be validated by checking the class identifier of the target object to make sure it falls into the target class' interval.

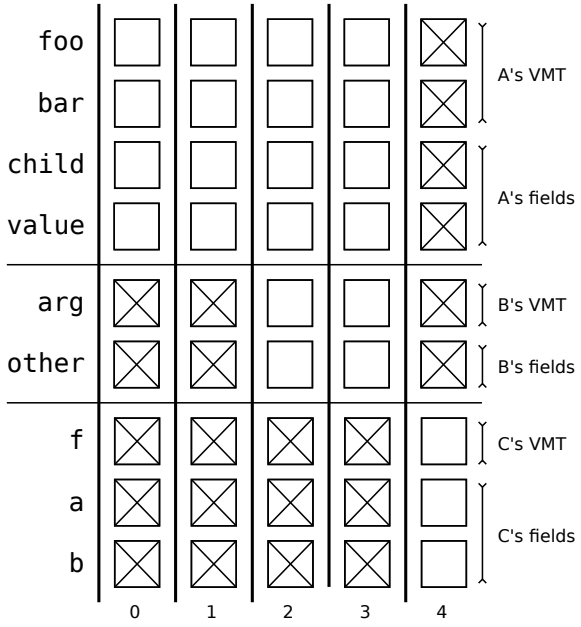

Figure 4.1. Array model.

Chapter 4

Arrays

Virgil's notions of arrays is similar to that of Java. The programmer can always check the length of an array by accessing the `length` field, and reads and writes are checked at runtime for being outside of the bounds of the array. What the programmer actually deals with are *references* to the arrays themselves; like objects, all fields, variables, and parameters which are typed in Virgil as an array are actually treated as references to arrays. These references are then passed by value in methods.

Array references normally would be implemented in C, again, via the use of pointers. Arrays would be converted to global variables, and pointers would in turn point to these global variables. Virgil takes an additional step by also including with the pointer a field indicating the length of the array. These two items would be wrapped into a `struct`, which would then be passed around. Of course, because of the same reasons why the traditional object representation doesn't work, this

strategy does not work either. Aside from the fact that we cannot use pointers, they cannot be included as members of `structs` either.

4.1 Array Grouping

Arrays in Virgil also fall under the rules for object allocation: arrays can only be created using the `new` operator with a constant-sized length inside constructors for classes or components. Once the initialization phase is complete, the compiler will know the total number of arrays in the program and their lengths. This information allows us to perform our array grouping technique.

In order to successfully represent references to arrays without pointers, we approach the problem in much the same way as with objects. We use the type information known during compilation to create canonical global variables for each type of an array, demarcated by the element type. All arrays which have the same element type are then grouped together and concatenated to form one long array. The original arrays as defined in the Virgil program are now subsets of this one canonical array.

This approach is illustrated in figure 4.1. The heap shown here consists of two arrays typed as `char[]`, two `int` arrays, and two arrays holding objects of different types, `A[]` and `C[]`. Although the classes `A` and `C` are incompatible, i.e. they do not intersect at all in the class hierarchy, we consider them both to be the same array type, as all objects are represented by one struct in C. The arrays are grouped and concatenated, forming one canonical array for each array type.

Array grouping accomplishes the prerequisites needed to successfully encode references to arrays without the need for pointers.

4.2 Array Referencing

With the array groups created, an array can be uniquely identified by its start offset, and its length. We solve this problem in much the same way as the object references; we create a single `struct` which holds the necessary information to identify an array. This array reference struct's basic structure is shown in figure 4.2. Because an array reference can be `null` like object references, we reserve a special flag for null. The other two fields indicate the start of the array within the canonical array, as well as the length of the array, respectively. The length information is used on all element accesses during runtime to ensure that accesses are valid and in the bounds of the array, similar to array accesses in Java or C#.

These array references are, like our object pseudopointer, passed by value into functions. Given an array reference and a expression evaluating to an index within the array, an element access would entail the following steps:

```
struct Array {
  char null;
  int start;
  int length;
}
```

Figure 4.2. Array reference `struct`.

```
method f(arr : int[], x : int) : int { return arr[x]; }

int f(struct Array arr, int x) { return int_array[arr.start + x]; }
```

1. Determine the overall index by summing the result of the index expression together with the `start` field of the array reference : `arr.start + x`.
2. The compiler determines the correct global array to access based on the static type of the array. All objects are considered to be one array type: `int_array`.
3. Perform the array access.

Although all array accesses are checked to ensure they are within bounds, because currently Virgil has no exception handling mechanism, the case where an invalid array access occurs will result in undefined behavior.

Array grouping allows us to determine the correct array to access during compile time based solely on the known type information of the array being accessed. Grouping all arrays of the same time to one canonical array eliminates the need for pointers, and allows us to represent references to arrays in a straightforward way with minimal performance overhead.

Chapter 5

Methods

The final area of major concern is virtual method dispatch without the use of tables or function pointers. The general approach for this is the use of *typecase* to examine the runtime type of an object in order to determine the correct function to invoke. To accomplish this, we create intermediate dispatcher functions. In general, each method call will be re-routed to a dispatcher function, which switches on the object's type identifier and then calls the correct function directly. This section will provide an overview of this approach.

5.1 Type Case for Method Dispatch

Our dispatcher method relies on several features of our object representation to successfully dispatch a call to the correct function: the runtime type information field, `TYPEID`, in the root class struct; the fact that the pointer offsets are ordered in such a way that the root class' component index will always be the first field in the pointer; and the fact that all classes are known during compile time in order to create a complete class hierarchy.

The dispatcher consists primarily of a switch statement on the `TYPEID` of the object, and calls the appropriate function for the class. If the method has a non-void return type, the dispatcher will return the result of the called function. One case statement will be placed for every class in the class hierarchy. We employ fall-through cases for classes that share a common method. An example of the dispatcher method for `foo` as defined in class `A` of our example hierarchy is shown in figure 5.1.

The dispatcher strategy incurs the highest performance penalty in our object representation model, since each method call will have to go through an intermediate dispatcher function before the correct code is executed. However, through static analysis and optimizations described later, we can skip the dispatcher for many method calls to which the destination method is definitely known during compile-time.

```
void Foo_dispatch(struct Pointer __this) {
  switch(Row_A[__this.f0]) {
    case 1: // B
      B_bar(__this);
      return;
    default:  // A
      A_bar(__this);
      return;
  }
}
```

Figure 5.1. Dispatcher function in C.

5.2 Delegates

Virgil supports the notion of delegates, which behave similarly to function pointers. Normally, they are compiled in C as function pointers, which then are passed around in the place of delegates. However, because AutoPilot does not support function pointers, we do not allow delegates in the Virgil programs at this time. It is possible to extend the concept of dispatchers and create a large dispatcher that switches between all functions that have the same signature, however we have not implemented this at this time. We therefore relegate the implementation of delegates to future work.

Chapter 6

Optimizations

The approaches we have just demonstrated represents the generalized model of how we implement the various language constructs of Virgil targeted for hardware compilation. However, we perform several additional optimizations to further reduce area needed and improve performance. Our optimizations fall into several categories, which will be discussed in this chapter.

6.1 Method Call Optimizations

Although all methods are virtual in Virgil, we again take advantage of the fact that all classes are known at compile time in order to replace some dispatcher calls with direct method calls. Some virtual methods may never be overridden by other classes in the class hierarchy; calls to these methods do not need a dispatcher, as it is known in complete certainty which method is being called. We are able to then remove the call to the dispatcher, instead replacing it with a direct call the function. This minimizes the impact to performance of using dispatchers.

In our example, the `arg` method in `B` and the `foo` method in `A` both exhibit the property of never being overridden by a child class. Calls to these functions elsewhere in the program, then, are replaced with direct calls to the C functions that represent the methods.

6.2 Bitwidth Optimization

Our compiler generates a variety of data structures and special integer values throughout the model presented. Special values such as the array lengths, object identifiers, array offsets, object table offsets, all are constant values generated during compilation time whose minimum and maximum values are known. Because of this fact, our compiler can optimize the sizes of the integer types needed to store these values.

AutoPilot allows for arbitrary bit-width integer types in C. It accomplishes this through various `typedef`s that hook into AutoPilot's own libraries. This feature is exposed in C via types such as `int6`, which represents a 6-bit signed integer. AutoPilot supports arbitrary bit integers from 1 bit to 1,024 bits. We take advantage of this feature to minimize the bit widths of all of the

compiler-generated data structures.

To illustrate this approach, let us use the heap layout as shown in figure 3.3. The pseudopointer consists of two components. Classes that use the first component are `A`, `B`, and `C`; the second component is used only by `B`. The widest row that is pointed to by the first component is the row for `A`, which consists of four elements. The row for `B`, on the other hand, only has two cells. Our compressed pseudopointer is then:

```
struct Ptr {
  uint1 null;
  uint2 comp1;
  uint1 comp2;
}
```

Our pseudopointer, as a result, is only 4 bits wide. One bit is required for the `null` flag, while 2 bits are needed to represent the last index in row `A`, `3`, and one bit to encode`1`, the last index in row `B`. This optimization, although minor, serves two purposes: it saves area to a small extent by reducing the bus widths; it also increases performance by a small amount by reducing the probability of wires of uneven lengths (due to routing).

Variables, fields, and parameters defined as type `int` within the Virgil program itself by the programmer are not modified. Because the compiler cannot determine the range of values that will be stored within these, their size will always be 32 bits. However, Virgil has another feature that allows the programmer to take advantage of this bitwidth optimization feature by using the raw types within Virgil. Raw types are unsigned types which can be defined as any number of bits between 1 and 64. Normally, we map these to one of the primitive unsigned integer types in C; however, when compiling with AutoPilot as the target, we can convert them into the exact number of bits that the programmer defined them to be.

6.3 Array and Object Initialization

One final required step that we must do is assign all the initialization data – the initial values of all arrays and objects — into a separate variables in C which are marked with `const`. AutoPilot recognizes this property and puts the data in a special ROM. Upon startup, this ROM data is copied into the RAM slots. This step comprises the *runtime initialization* phase when the hardware is executed. The overhead introduced by this operation is measured in the benchmarks section; it depends primarily on the amount of data that needs to be copied from ROM to RAM.

Chapter 7

Experimental Results

7.1 Benchmarks

We wrote four Virgil benchmark applications to measure and compare the performance of our compiler. Three were translations of cryptographic benchmark programs authored in C found in the CHStone benchmark suite [6], a well-known suite benchmark programs written in C designed to be synthesized to FPGAs. These benchmarks, because they are translated from C, a non-object oriented language, lack the use of most of the object-oriented features that are available in Virgil. They primarily showcase the raw computational veracity of the compiler. To make up for this deficiency, a fourth benchmark, the Richards benchmark, was also translated into Virgil [14]. The Richards benchmark simulates the task dispatcher in the kernel of an operating system, and was translated from an object-oriented Java/C++ implementation. The Richards benchmark does little in the way of raw computation, but exercises many features of the language that were not covered by our cryptographic benchmarks. Richards uses many Virtual methods which require dispatchers, and there is a nontrivial amount of objects allocated during the initialization phase. We chose Richards to enable comparison of Virgil and C++.

The following list describes in more detail the various benchmarks that we have chosen to use. They have been specifically chosen because they do not use any floating point arithmetic, as Virgil does not have floating-point primitive types. The first three in the list are our cryptographic benchmarks from the CHStone suite. The last one is the object-oriented Richards benchmark.

- **AES** — An implementation of the Advanced Encryption Standard, a popular and modern encryption cipher.
- **Blowfish** – An implementation of the Blowfish block cipher algorithm.
- **SHA** – An implementation of the cryptographic hash function, SHA-1.
- **Richards** – Simulation of a task-dispatcher component of an operating system kernel.

Figure 7.1 shows the various sizes of each benchmark program in line numbers for both the original source code and our Virgil translation.

	Lines of code	
	Original	Virgil
Originally in C:		
AES	791	669
Blowfish	1320	1548
SHA	1349	1187
Originally in C++:		
Richards	705	437

Figure 7.1. Benchmark code length.

	CPU (xeon)		**CPU (atom)**		**FPGA**				
Benchmark	Time (us)	Energy (mJ)	Time (us)	Energy (mJ)	Time (us)	Energy (mJ)	Slices	FlipFlops	BRAM
AES									
C	23	1.9	92	0.37	34	0.04	4,803	6,641	54
Virgil/wide	83	6.7	317	1.27	103	0.14	6,199	8,198	51
Virgil/hybrid	85	6.8	317	1.27	106	0.14	6,575	8,253	51
Blowfish									
C	222	17.7	834	3.34	1,144	1.52	6,795	8,962	63
Virgil/wide	877	70.2	1,786	7.15	2,092	2.74	4,689	6,031	69
Virgil/hybrid	889	71.1	2,587	10.35	2,040	2.65	4,700	6,029	69
SHA1									
C	319	25.4	1,093	4.37	1,565	2.07	5,715	8,409	65
Virgil/wide	1,070	85.6	2,131	8.52	1,525	1.98	4,858	6,595	64
Virgil/hybrid	1,074	85.9	2,630	10.52	1,525	2.04	4,890	6,598	64
Richards									
C++	10,065	805.2	39,900	159.60	N/A	N/A	N/A	N/A	N/A
Virgil/wide	11,164	893.1	36,331	145.32	16,065	21.21	4,330	5,519	68
Virgil/hybrid	29,135	2,330.8	61,622	246.49	14,433	18.91	4,317	5,355	67

Figure 7.2. Experimental results.

7.2 Platform

The performance of our compiler was compared with that of two different CPUs: an Intel Xeon CPU, which is Intel's high-performance CPU offering, and an Intel Atom CPU, which is the low-power mobile offering. The Xeon's heftiness offers speed at a cost of power consumption, while the Atom's leanness ensures that its power consumption will be very low compared to the Xeon,

sacrificing computational performance. The Xeon processor gives us an idea of the computational veracity of our benchmark applications that can be achieved on ordinary, off-the-shelf server and workstation machines, while the Atom gives us a better idea of how the lowest-power x86 CPUs compare to the FPGA in terms of both performance and power consumption. The Xeon E5430, upon which our Xeon platform is based, has a TDP (Thermal design power) of 80 Watts [9]. On the other hand, our Atom 230 CPU boasts a TDP of only 4 Watts [8]. Because the Atom's target market overlaps significantly with that of FPGAs, we believe that the Atom will be a more interesting comparison than the Xeon.

The *TDP* of a processor indicates its maximum designed power consumption. We use these figures in our estimates of the power consumed when our benchmark applications are executed on the respective CPUs. While additional power is consumed by the support devices needed to facilitate a CPU — memory, storage drives, and other peripheral devices — we are only interested in the power directly consumed by the CPU in this paper. FPGA power consumption can be measured in a more accurate way; each FPGA vendor usually provides a tool that can precisely estimate the amount of power consumed by a particular design when it is turned on. Both of these figures, the TDP of a CPU and the estimated power consumption of an FPGA in watts, which can be multiplied by the execution time to give a estimated figure of the power consumed by the program in joules.

CPU benchmarks were performed via an x86_64 `gcc` compiler running on Ubuntu Linux, kernel 2.6.32. The benchmarks were run 200,000 times, with the overall run time being divided by 200,000 to obtain the average per-execution time. FPGA measurement was done via the `GHDL` VHDL compiler, which converts FGPAs designs into an x86 program, which can be run from within Linux. This gave us the ability to have a better view of the internal timings and signal data. The simulation results were confirmed by re-executing the design on an Xilinx Virtex-II FPGA chip. The FPGA vendor (Xilinx) tools were used for synthesis and layout.

The Intel Xeon E5430 processor contains 4 cores, each running at 2.66 GHz. It has a 6 MB shared cache. The system was further equipped with 32 GB of DDR2 RAM, although the benchmark programs never used more than 500MB of memory. The benchmarks were compiled using GCC 4.4.3 on Ubuntu "Lucid Lynx" 10.04.

The Intel Atom 230 is a single-core, hyper-threaded CPU running at 1.6 GHz with 512KB of cache. 1 GB of DDR2 memory is also installed on the system. Again, the benchmarks were compiled using GCC 4.4.3 on Ubuntu "Lucid Lynx" 10.04.

The FPGA simulation platform is primarily the GHDL VHDL compiler version 0.28dev running, again, on Ubuntu Linux 10.04. The hardware specifications of the simulation system are not important, as the simulator executes the design at a specified clock speed (100MHz in our case) regardless of the underlying hardware.

Benchmark	Initialization (us)
AES	23us
Blowfish	231us
SHA1	330us
Richards	2us

Figure 7.3. Runtime initialization periods.

7.3 Measurements

Figure 7.2 show our measurements. On each execution platform, the original C benchmark was first compiled and executed to establish "original" performance. Our compiler then compiles the benchmark programs in two specific modes for comparison: first, the notion of the "uncompressed" vertical object model, which contains wasted space, and our "hybrid" object model which was described in this paper. The uncompressed model is measured in order to show that our table compression and object reference representation does not have an adverse effect on the overall performance when run on an FPGA, although it may result in slower run times on the CPU. Finally, area measurements were gathered from the synthesis reports generated by the Xilinx synthesis tool. We report three numbers for area usage:

1. **Slices** – a quantitative measurement which represents the size of the logic of the design.

2. **Flip-flops** – Flip-flops are the on-chip ROM, which reflects the amount of read-only memory needed by the design.

3. **BRAM units** – On-chip RAM is split into subunits called BRAMS.

7.3.1 Runtime Initialization

As discussed in the optimization section, an additional step that must be taken is the *runtime initialization* phase. This step is a one-time operation that occurs at the beginning of execution that copies all the initialization data stored in the ROM into the RAM that takes up a nontrivial amount of time. The table in figure 7.3 shows the time taken for this required initialization phase for the different benchmarks. This number was gathered during simulation of the FPGA design.

These numbers are already included in the performance numbers included in figure 7.2. However, the amount of time spent purely on the logic of the Virgil benchmarks can be obtained by subtracting this number from the total execution time.

7.4 Microbenchmarks

The scalability factor of our object model was also tested in order to ensure that it supports large programs with many objects and/or deep class hierarchies. Two benchmarks were written: one to test the effect of the dispatcher on large class hierarchies, and the other to test the the impact of large amounts of objects upon method calls. For the former, class hierarchies of three, six, and twelve were benchmarked, and for the latter, programs consisting of one class and ten, one hundred, and one thousand objects were measured. In all cases, one million method calls were issued.

We found that there was no significant difference in performance between the various benchmarks; all performed within 3-5% of each other. This strongly indicates that there should be minimal scalability issues with large programs and our object model.

7.5 Assessment

We approached the performance assessment from three different perspectives: energy, run time, and physical size. Primarily, we were interested in the amount of energy consumed by the various designs during execution. Admittedly, energy savings may be less attractive if the run time were to degrade significantly when switched to an FPGA. Thus, secondly, we performed overall measurements of run time across platforms. Thirdly, we also analyzed the physical size of the hardware designs created by our compiler, as fabrication cost for an FPGA chip is primarily driven by the physical size of the design.

Time and Energy. As can be expected, the Xeon platform is the fastest and uses the most energy. The Xeon processor executed the benchmarks 2 to 4 times faster than the Atom processor, but consumed 5 to 10 times more energy doing it.

For the three cryptographic benchmarks, C on the Atom processor executed 1.2 to 2.4 times faster than Virgil on the FPGA, but consumed 1.3 to 2.6 times more the energy doing it. In contrast, for the object-oriented Richards benchmark, the FPGA is better in both dimensions: C++ on the Atom processor executed 2.8 times *slower* than Virgil on the FPGA, and consumed 8.4 times more energy doing it. Thus, for the object-oriented benchmark we can get the best of both worlds by switching from C++ on Atom to Virgil on the FPGA: faster execution and almost an order of magnitude energy savings.

Remarkably, Richards in C++ on Xeon executes within 2x faster than Virgil on the FPGA, but consumes more than 42x more energy doing it.

For the Richards benchmark, when we compare Virgil/wide to Virgil/hybrid, we see a big jump in run time and energy consumption for both Xeon and Atom, but a significant drop for the FPGA. The reason is that the hybrid object model is a better fit for the FPGA than the wide object model, but is wasteful on a CPU.

For SHA1, the execution time and energy consumption of Virgil on the FPGA is even lower than of C on the FPGA.

If initialization time is omitted from the result, then the AES benchmark executes slightly faster in Virgil on the FPGA than in C on the Xeon! For an informal comparison, Huang et al. reported that DES on an FPGA ran 14 times slower than DES on a Core 2 Duo processor with a frequency of 2.66 GHz. We believe that our Virgil compiler is a significant improvement over the previous work on marrying object-oriented programming paradigms with hardware synthesis toolchains.

Physical Size. For each of the three cryptographic benchmarks, we can compare the slices, FlipFlops, and BRAM in C and in Virgil. We find that our area usage numbers are comparable. We have managed to occupy less area than that of even the original C program's design in two out of three cases. In both of these cases, our version of the benchmark occupied significantly less area. The AES benchmark, however, occupied a similar margin more area.

The interpretation of the Richards benchmark results must be approached differently. Because the original program cannot be synthesized through AutoPilot as it uses many of the language constructs, especially pointers, in such a way that is disallowed by AutoPilot, we do not have a comparison on the FPGA for the equivalent non-Virgil version. Furthermore, Virgil provides a way to write the Richards benchmark that would be impossible otherwise to do (short of replicating our model by hand in C). Therefore, our sole comparison available is the execution time of the C++ version running on the CPUs. It is also the only benchmark in which we have a significant number of objects and classes; the computational CHStone benchmarks are not implemented in a object-oriented fashion. Here, we see the advantages of our object compression scheme. We used a smaller number of both flip-flops and BRAMs, as well as gain a significant performance boost over the non-compressed version.

The Virgil Compiler. The measurements show significant variation across the cryptographic benchmarks. For example, AES in C on the Atom is 1.2 times faster than in Virgil on the FPGA, while Blowfish in C on the Atom is 2.4 times faster than in Virgil on the FPGA. It is difficult to pinpoint the exact causes of the performance difference because optimizations are done at multiple points through the compilation process: our Virgil compiler attempts to produce optimized C code, the AutoPilot tool itself aims to emit optimized HDL design code, and the final FPGA vendor-specific synthesis tool aims to produce an optimal physical layout of the design on the FPGA chip, so In future work we will investigate further how to optimize C code for AutoPilot.

Chapter 8

Conclusion

The compiler that we have introduced, which enables a full tool chain that leads from high-level object-oriented Virgil code to low-level VHDL designs, allows software engineers to easily reap the enormous energy consumption benefits that FPGAs have to offer while still exhibiting reasonable performance and competitive area.

This system is still a preliminary investigation, and much work can be done to further improve the experience of the programmer. In particular, work can be done to better extend Virgil to some domains of hardware programming is currently out of reach. Examples of such domains are that of streaming data models, floating-point arithmetic, and designs that interact with external hardware. All three of these can be achieved elegantly with well-designed extensions to Virgil. Further static analysis, or an explicit modifier to make certain arrays and objects read-only, would help to shorten runtime initialization by reducing the amount of data that must be copied from ROM to RAM.

Furthermore, improvements can be made to overall programmer experience in terms of designing programs with the paradigms which already exist. A big notion that Virgil is currently lacking is support for user exceptions; currently, any attempts at dereferencing a null reference or accessing an array out of bounds leads to undefined behavior. By implementing an exception system as well as a way in hardware to handle these exceptions would greatly ease the task of designing hardware that gracefully exits when error conditions are encountered.

Finally, several features currently allowed in Virgil are not supported in our representation. These include:

- Recursion
- Delegates (function pointers)
- Generics

The support for these features, along with the various improvements that could be made listed above, would make Virgil a truly powerful platform on which to program FGPAs.

References

[1] David Baldwin. Structured ASIC challenges FPGA. *Nikkei Electronics Asia*, September 2003.

[2] Jayaram Bhasker. *A Verilog HDL Primer*. Star Galaxy Publishing, 2005.

[3] David C. Black, Jack Donovan, Bill Bunton, and Anna Keist. *SystemC: From the Ground Up*. Springer, 2004.

[4] Craig Chambers. *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1992.

[5] J. Cong and Y. Zou. Lithographic aerial image simulation with FPGA-based hardware acceleration. In *In FPGA'08, Proceedings of 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 20–29, 2008.

[6] Yuko Hara, Hiroyuki Tomiyama, Shinya Honda, and Hiroaki Takada. Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis. *Journal of Information Processing*, 17:242–254, 2009.

[7] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric M. Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *Proceedings of ECOOP'08, European Conference on Object-Oriented Programming*, pages 76–103, 2008.

[8] Intel. Atom processor 230. Information available from http://ark.intel.com/Product.aspx?id=35635.

[9] Intel. Xeon processor E5430. Information available from http://processorfinder.intel.com/details.aspx?sSpec=SLANU.

[10] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. In *In FPGA'06, Proceedings of 14th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pages 21–30, 2006.

[11] Chris Lattner. LLVM: An infrastructure for multi-stage optimization. Master's thesis, University of Illinois at Urbana-Champaign, 2004.

[12] Oskar Mencer, Marco Platzner, Martin Morf, and Michael J. Flynn. Object-oriented domain specific compilers for programming FPGAs. *IEEE Transactions on VLSI Systems*, 9(1):205–210, 2001.

[13] Volnei A. Pedroni. *Circuit Design with VHDL*. MIT Press, 2004.

[14] Martin Richards. Benchmarking with the Richards benchmark. http://research.sun.com/people/mario/ java_benchmarking/richards/richards.html.

[15] Martin Schoeberl. Java technology in an FPGA. In *In FPL'04, Proceedings of the International Conference on Field-Programmable Logic and its Applications*, 2004.

[16] Ben L. Titzer. Virgil: Objects on the head of a pin. In *Proceedings of OOPSLA'06, ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, 2006.

[17] Ben L. Titzer, Joshua Auerbach, David F. Bacon, and Jens Palsberg. The ExoVM system for automatic VM and application reduction. In *Proceedings of PLDI'07, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 352–362, San Diego, California, June 2007.

[18] Ben L. Titzer and Jens Palsberg. Vertical object layout and compression for fixed heaps. In *Proceedings of CASES'07, International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 170–178, Salzburg, Austria, September 2007. A revised version of the paper appeared in Semantics and Algebraic Specification, Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday, Springer, LNCS 5700, 2009.

[19] Z. Zhang, Y. Fan, W. Jiang, G. Han, C. Yang, and J. Cong. Autopilot: A platform-based ESL synthesis system. In P. Coussy and A. Morawiec, editors, *High-Level Synthesis: From Algorithm to Digital Circuit*. Springer Publishers, 2008.

DISTRIBUTION:

1 Stephen Kou
10500 National Blvd. #43
Los Angeles, California 90034

1	MS 0576	John Feddema, 05511
1	MS 0899	Technical Library, 9536 (electronic copy)
1	MS 0359	D. Chavez, LDRD Office, 1911

Sandia National Laboratories