Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary

Shan Shan Huang[1], Amir Hormati[2], David F. Bacon[3], and Rodric Rabbah[3]

[1] Georgia Institute of Technology
[2] University of Michigan
[3] IBM Research

Abstract. The paradigm shift in processor design from monolithic processors to multicore has renewed interest in programming models that facilitate parallelism. While multicores are here today, the future is likely to witness architectures that use reconfigurable fabrics (FPGAs) as co-processors. FPGAs provide an unmatched ability to tailor their circuitry per application, leading to better performance at lower power. Unfortunately, the skills required to program FPGAs are beyond the expertise of skilled software programmers. This paper shows how to bridge the gap between programming software vs. hardware. We introduce Lime, a new Object-Oriented language that can be compiled for the JVM or into a synthesizable hardware description language. Lime extends Java with features that provide a way to carry OO concepts into efficient hardware. We detail an end-to-end system from the language down to hardware synthesis and demonstrate a Lime program running on both a conventional processor and in an FPGA.

1 Introduction

The end of the free ride from clock scaling has stimulated renewed interest in alternative computer architectures. Due to the increased complexity of these architectures, there has also been a corresponding revival of interest in alternative models for programming them.

Most of the attention has been focused on multicore chips, but many other types of systems are being produced and explored: SIMD, graphics processors, "manycore", and reconfigurable hardware fabrics. While multicore chips are the most straightforward for chip manufacturers to produce, it remains an open question as to which hardware organization is the most efficient or the easiest to program. Furthermore, as power outweighs chip area, it seems likely that systems will become increasingly heterogeneous.

Among these alternative architectures, reconfigurable fabrics such as field-programmable gate arrays (FPGAs) have many compelling features: low power consumption, extremely high performance for many applications, a high degree of determinism, and enormous flexibility. Because FPGAs route and operate on single bits, it is possible to exploit many different kinds of parallelism either

individually or in combination: at the micro-scale of bits or the macro-scale of tasks, with pipelining or data parallelism, etc.

Recently, chip manufacturers have begun providing interfaces to allow the kinds of high-bandwidth data transfer that makes it easier to connect accelerator chips to CPUs (for instance, AMD's Torenza and Intel's QuickAssist). Some motherboards come with an open socket connected to such a bus into which one can plug an FPGA. The increasing availability of systems with FPGAs offers an opportunity to customize processing architectures according to the applications they run. An application-customized architecture can offer extremely high performance with very low power compared to more general purpose designs.

However, FPGAs are notoriously difficult to program, and are generally programmed using hardware description languages like VHDL and Verilog. Such languages lack many of the software engineering and abstraction facilities that we take for granted in modern Object-Oriented (OO) languages. On the other hand, they do provide abstractions of time and a much more rigorous style of modular decomposition. In hybrid CPU/FPGA systems, additional complexity is introduced by the fact that the CPU and the FPGA are programmed in completely different languages with very different semantics.

The goal of the Liquid Metal project at IBM Research is to allow such hybrid systems to be programmed in a single high-level OO language that maps well to both CPUs and FPGAs. This language, which is backward-compatible with Java, is called *Lime*.

While at first glance it may seem that conflicting requirements for programming these different kinds of systems create an inevitable tension that will result in a hodgepodge language design, it is our belief that when the features are provided at a sufficiently high level of abstraction, many of them turn out to be highly beneficial in both environments.

By using a single language we open up the opportunity to hide the complexity of domain crossing between CPU and FPGA. Furthermore, we can fluidly move computations back and forth between the two types of computational devices, choosing to execute them where they are most efficient or where we have the most available resources.

Our long-term goal is to "JIT the hardware" – to dynamically select methods or tasks for compilation to hardware, potentially taking advantage of dynamic information in the same way that multi-level JIT compilers do today for software. However, many challenges remain before this vision can be realized.

In this paper, we present an end-to-end system from language design to co-execution on hardware and software. While some of the individual components are incomplete, significant portions of each part of the system have been built, and the overall system architecture is complete.

The system that we present in this paper consists of the components shown in Figure 1 (the components are labeled with the paper sections in which they are described). The system consists of a front-end for the Lime language which can generate either Java bytecodes or a spatial intermediate language suitable for computing on FPGAs. When compiling to hardware, a sequence of compilation

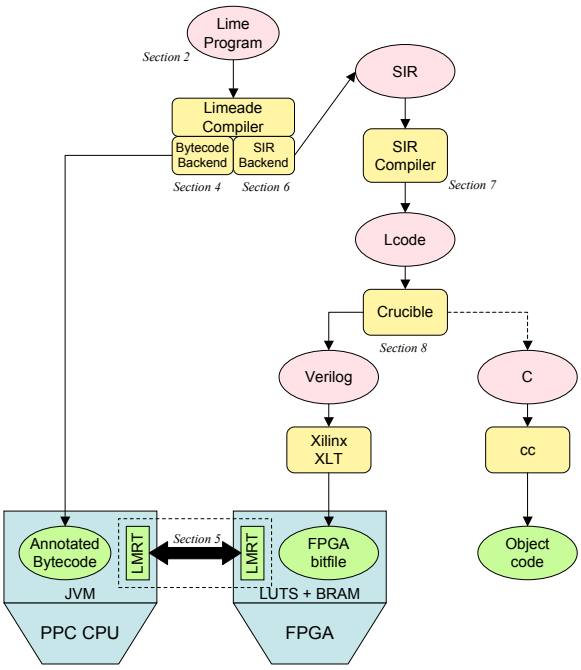

Fig. 1. The Liquid Metal Compilation and Runtime System.

steps is used to produce *bitfiles* which can be loaded onto the FPGA. The Liquid Metal Runtime system (LMRT) consists of portions that reside on both the FPGA and on the CPU. LMRT handles communication and synchronization between the two domains, as well as loading of FPGA code.

The infrastructure also allows generation of C code, which could be used for compilation to either standard multicores or SIMD processors like Cell. However, we have not investigated the performance of this portion of the tool chain, so we do not discuss it in this paper.

2 Lime: Language Design

Lime is designed with two goals in mind: Programmers should be able to program with high-level OO features and abstractions; These high-level programs should be amenable to bit-level analysis and should expose parallelism. To achieve these goals, Lime extends Java with value types, value generic types, and `enum`-indexed arrays. In this section, we discuss these features and demonstrate how they can be used by the programmer. We will also highlight their implications for the compiler, particularly with respect to efficient synthesis to an FPGA.

A value type in Lime can be an `enum`, class, or interface, annotated with the modifier `value`. The defining characteristic of value types is that they are immutable. An object of a value type, once created, never changes. We begin our exposition with the building block of value types: value `enum`'s.

Lime's value types share many properties with those of Kava [1]. However, they have simpler type rules, can be safely used in Java libraries, and support generics.

2.1 Value Enumeration Types

A value `enum` is a restricted form of the Java `enum` type. It represents a value type with a bounded number of possible values. The following is a user-defined representation of type `bit`, with two possible values, `bit.zero`, and `bit.one`:

```
public value enum bit { zero, one; }
```

Unlike Java `enum`'s, a value `enum` cannot define non-default constructors (i.e. constructors taking arguments), nor can it contain mutable fields. We elaborate in Section 2.5 the type checking performed on all value types to ensure their immutability. For the moment, it is sufficient to know that all fields of a value type, including `enum`'s, must be `final` references to objects of value types.

The Lime compiler provides a number of conveniences for value `enum`'s, making them as easy to use as values of primitive types.

Default Values. A variable of a value `enum` type is never `null`. If uninitialized, the variable is assigned a default value: the first value defined for that `enum` type. For example, in declaration `bit b;` variable `b` has the default value `bit.zero`. In fact, a default value is automatically given for *all* value types, as we will show shortly.

Compiler-Defined Operators. Arithmetic operators such as `+`, `-`, `++`, and `--`, are automatically defined for value `enum`'s. For example, `bit.one + bit.zero` returns `bit.one`. Similarly, `bit.one + bit.one`, returns `bit.zero`, akin to how integers wrap around when overflowing their range. Comparison operators are also defined for value `enum`'s. For example, `bit.one > bit.zero` returns `true`.

Lime also provides programmers an easy way to produce and iterate over a range of values. Lime introduces the binary operator, `::`. The expression `x :: y` produces an object of `lime.lang.Range<T>` or `lime.lang.ReverseRange<T>`, depending on whether `x < y`, or `x > y`, and where `T` is the least upper bound type of `x` and `y`. Both `Range<T>` and `ReverseRange<T>` are value types, implementing the `Iterable<T>` interface in Java. They are thus usable in the "for-each" style loops introduced since Java 5. For example, the following code defines a loop over the range of values greater than or equal to `bit.zero`, and less than or equal to `bit.one`:

```
for ( bit b : bit.zero :: bit.one ) { System.out.println(b); }
```

Furthermore, there is nothing required of the programmer to indicate whether a range is ascending or descending, even when the operands' values cannot be statically determined. For instance, in the code below, depending on the arguments used to invoke `printBits`, the object generated by `begin :: end` can be either `Range<bit>` or `ReverseRange<bit>`:

```
void printBits(bit begin, bit end) {
   for ( bit b : begin :: end ) { System.out.println(b); }
}
```

In Lime programs, programmers often need to iterate over the entire range of possible values of a value `enum`. A convenient shorthand is provided for iterating over this range. For example, `for (bit b) {...}` is equivalent to `for (bit b : bit.first :: bit.last) {...}`. Such a default range is always an ascending one.

The `::` operator is automatically defined for any value type supporting the operators `++`, `--`, `<` and `>`. Lime also supports the `::` operator for Java's primitive integral types, such as `int`, `short`, etc.

Compiler-Defined Fields and Methods. In addition to operators, static fields `first` and `last` are automatically defined to reference the smallest and largest values in a value `enum`'s range. For instance, `bit.first` returns `bit.zero`. These fields may seem redundant for a known `enum` type, but they become invaluable when we iterate over the range of an `enum` type *variable*, where the exact values of an `enum` are not known statically.

Methods `next()` and `prev()` are generated to return the value proceeding and preceding the value invoking the method: `bit.first.next()` returns `bit.one`.

Since objects of value `enum`'s (and in fact, all value types) do not have object identity at the Lime language level (i.e., all instances of `bit.zero` should be treated as the same object), the Lime compiler automatically generates `equals(Object o)` and `hashCode()` methods for these `enum`'s. The compiler also overloads the `==` operator for value `enum`'s to invoke `equals(Object o)`. (An exception to this case is when `==` is used inside the definition of `equals(Object o)` itself.) Note that this is exactly the opposite from what is in Java: the `equals(Object o)` method in Java defaults to invoking `==` and comparing object identity.

User-defined Operators. Lime also allows programmers to define their custom operators, or even override the automatically defined ones. For instance, we can define a unary complement operator for `bit`:

```
public bit ~ this { return this++; }
```

A binary operator could be similarly defined. For instance, the operator `&` for `bit` can be defined as follows:

```
public bit this & (bit that) { return this == one && that == one; }
```

Operator definitions are converted into compiler-generated methods, dispatched off of the `this` operand. For example, the $\sim$ operator definition becomes: `public bit $COMP() { return this++; }`, and the definition for `&` thus becomes `public bit $AND(bit that) { ... }`.

2.2 `enum`-indexed Arrays

Lime also extends Java with `enum`-indexed arrays. For example, `int[bit] twoInts;` declares an `int` array, named `twoInts`. The size of `twoInts` is bounded by the number of values in the value `enum` type `bit`. Thus, `twoInts` has a fixed size of 2. Furthermore, only an object of the array size's `enum` type can be used to index into an `enum`-indexed array. The following code demonstrates the use of `enum`-indexed arrays.

```
int i = twoInts[0];        // ILLEGAL! 0 is not of type bit
int j = twoInts[bit.zero]  // OK
```

An `enum`-indexed array has space automatically allocated for it by the compiler. `enum`-indexed arrays provide a nice way to express fixed-size arrays, where the type system can easily guarantee that array indexes can never be out of bounds. Both of these are important properties for laying out the program in hardware – but are also valuable for writing exception-free software and for compiling it for efficient execution.

2.3 A More Complex Value Type: `Unsigned`

Using value `enum`'s and `enum`-indexed arrays, we can now define a much more interesting value class, `Unsigned32`:

```
public value enum thirtytwo { b0,b1,...,b31; }

public value class Unsigned32 {
   bit data[thirtytwo];

   public Unsigned32(bit vector[thirtytwo]) { data = vector; }

   public boolean this ^ (Unsigned32 that) {
      bit newData[thirtytwo];
      for ( thirtytwo i)
         newData[i] = this.data[i] ^ that.data[i];
      return new Unsigned32(newData);
   }
   ... // define other operators and methods.
}
```

`Unsigned32` is an OO representation of a 32-bit unsigned integer. It uses the value `enum` type `thirtytwo` to create an `enum`-indexed array of exactly 32 `bit`'s, holding the data for the integer. The definition of `Unsigned32` exposes another interesting feature of the Lime compiler. Recall that a value type must have a default value that is assigned to uninitialized variables of that type. This means each value type must provide a default constructor for this purpose. Notice however that there is no such default constructor defined for `Unsigned32`. Conveniently, the Lime compiler can automatically generate this constructor for value types. The generated constructor initializes each field to its default value. Recall that one of the typing requirements of value types is that all fields must be references to value types. Thus, each field must also have a default value constructor defined (or generated) for it. The base case of our recursive argument ends with value `enum`'s. Thus, it is always possible for the Lime compiler to generate a default constructor.

Implications for the Compiler. Even though `Unsigned32` is defined using high-level abstractions, the combination of value `enum`'s and `enum`-indexed arrays exposes it to bit-level analysis. We can easily analyze the code to see that an object of `Unsigned32` requires exactly 32 bits: each element of `data[thirtytwo]` is of type `bit`, which requires exactly 1 bit; there are 32 of them in `data[thirtytwo]`.

This high level abstraction provides the Lime compiler with a lot of flexibility in both software and hardware representations of a value object and its operations. In software, Lime programs can be compiled down to regular Java bytecode and run on a virtual machine. We can choose to represent objects of `Unsigned32` and `thirtytwo` as

true objects, and iterations over values of `thirtytwo` are done through `next()` method calls on the iterator. However, without any optimizations, this would yield very poor performance compared to operations on a primitive `int`. We can thus also choose to use the bit-level information, and the knowledge that value objects do not have mutable state, to perform optimizations such as semantic expansions [2]. Using semantic expansions, value objects are treated like primitive types, represented in unboxed formats. Method invocations are treated as static procedure calls. These choices can be made completely transparent to the programmer.

The same analogy holds for hardware. Existing hardware description languages such as VHDL [3] and SystemC [4] require programmers to provide detailed data layouts for registers, down to the meaning of each bit. In contrast, Lime's high level abstraction allows the compiler to be very flexible with the way object data is represented in hardware. For instance, in order to perform "dynamic dispatch" in hardware, each object must carry its own type information in the form of a type id number. However, we can also strip an object of its type information when all target methods can be statically determined, and achieve space savings. The hardware layout choices are again transparent to the programmer.

The definition of `Unsigned32` exposes bit-level parallelism when it is natural to program at that level. Even more performance speed up can be gained through coarser-grained parallelism, where entire blocks of code are executed in a parallel or pipelined fashion. Very sophisticated algorithms have been developed to discover loop dependencies and identify which loops can be parallelized or pipelined safely. The knowledge of immutable objects make Lime programs even more amenable to these techniques. Our eventual goal is to design language constructs that promote a style of programming where different forms of parallelism are easily discovered and easily exploited.

2.4 Generic Value Types

A closer inspection of `Unsigned32` shows that its code is entirely parametric to the value `enum` type used to represent the length of the array `data`. No matter what `enum` is used to size `data`, the definitions for the constructor and operator `^` are exactly the same, modulo the substitution of a different `enum` for `thirtytwo`. A good programming abstraction mechanism should allow us to define these operations once in a generic way. Lime extends the type genericity mechanism in Java to offer exactly this type of abstraction. The following is a generic definition of `Unsigned<W>`, where type parameter `W` can be instantiated with different value `enum` to represent integers of various bit width:

```
public value class Unsigned<W extends Enum<W>> {
   bit data[W];

   public Unsigned(bit vector[W]) { data = vector; }

   public Unsigned<T> this ^  (Unsigned<T> that) {
      bit newData[T];
      for ( T i )
          newData[i] = this.data[i] ^ that.data[i];
      return new Unsigned<T>(newData);
   }
   ... // similarly parameterize operator definitions
}
```

Thus, to represent a 32-bit integer, we simply use type `Unsigned<thirtytwo>`. Similarly, we could use `Unsigned<sixtyfour>` to represent a 64-bit integer where `sixtyfour` is defined as follows:

```
public value enum sixtyfour { b0,b1,...,b63; }
```

Note that type parameters to value type are assumed to be value types, and can only be instantiated with value types.

For notational convenience, Lime offers a limited form of type aliasing. A `typedef` declaration can appear wherever variable declarations are allowed, and are similarly scoped. For example, the following statement declares `Unsigned32` as an alias for `Unsigned<thirtytwo>`:

```
typedef Unsigned32 = Unsigned<thirtytwo>;
```

We use the aliased forms of the `Unsigned<W>` class for the remainder of the paper.

2.5 Type-checking Value Types

In order to ensure that the objects of value types are truly immutable, we must impose the following rules on the definition of a value type:

1. A field of a value type must be `final`, and of a value type. The keyword `final` is assumed in the definition of value types and is inserted by the Lime compiler. Compile-time checks make sure that assignment to fields only happen in initializers.
2. The supertypes of a value type must also be value types (with the exception of `Object`).
3. The type parameter of a value type is assumed to be a value type during the type checking process, and can only be instantiated by value types.
4. Objects of value types can only be assigned to variables of value types.

The first three rules are fairly straight forward. The last rule requires a bit more elaboration. The Lime compiler imposes that value types can only be subtypes of other value types, except for `Object`. Therefore, the only legal assignment from a value type to a non-value type is an assignment to `Object`. In this case, we "box" the object of value type into an object of `lime.lang.BoxedValue`, a special Lime compiler class. The boxed value can then be used as a regular object. In fact, this is the technique used when a value type is used in `synchronized`, or when `wait()` is invoked on it.

Method `equals(Object o)` requires special treatment by these rules. The `equals` method must take an argument of `Object` type. It is inefficient to box up a value type to pass into the `equals` of another value type, which then has to strip the boxed value before comparison. Thus, the Lime compiler allows a value type to be passed into the `equals` of value types without being boxed. These `equals` methods have been type-checked to ensure that they do not mutate fields, it is thus safe to do so.

It is also important to point out that an array holding objects of value types is not a value type itself. Neither is an `enum`-indexed array holding objects of value types. The contents of the array can still mutate. A value array, then, is expressed as `(value int[]) valInts`. Similarly for value `enum`-indexed arrays.

A value array must be initialized when it is declared. All further writing into the array is disallowed. Our syntax does not allow multiple levels of immutability in arrays.

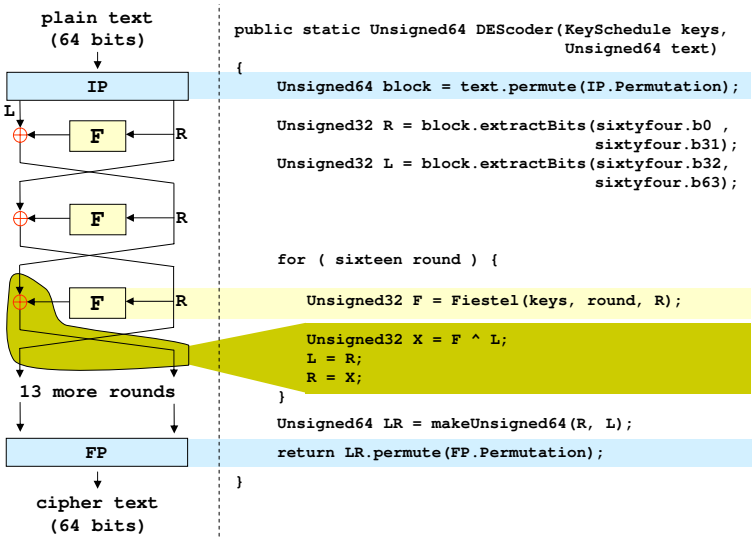

Fig. 2. Block level diagram of DES and Lime code snippet.

It is not possible to express a mutable array of value arrays, for example. The `value` keyword at the outside means the entire array, at all dimensions, are immutable.

Finally, methods `finalize()`, `notify()`, and `notifyAll()` can never be called on objects of value types. Objects of value types have no storage identity, thus these methods do not make sense for value objects.

3 Running Example

The Liquid Metal system is somewhat complex, consisting of a front-end compiler that generates bytecode or an FPGA-oriented spatial intermediate representation (SIR), a high-level SIR compiler, a layout planner, a low-level compiler, and finally a synthesis tool. In order to demonstrate how all of these components fit together, we will use a single running example throughout the rest of the paper.

Our example program implements the Data Encryption Standard (DES). The program inputs plain text as 64-bit blocks and generates encrypted blocks (cipher text) of the same length through a series of transformations. The organization of the DES algorithm and its top level implementation in Lime are shown in Figure 2. The transformations occur in 16 identical rounds, each of which encrypts the input block using an encryption key. The plain text undergoes an initial permutation (`IP`) of the bit-sequence before the first round. Similarly, the bit-sequence produced in the final round is permuted using a final permutation (`FP`). The output of the initial permutation is partitioned into two 32-bit half blocks. One half (`R`) is transformed using a `Feistel` function. The result of the function is then exclusive-OR'ed (`xor`) with the other half (`L`). The two halves are then interchanged and another round of transformations occurs.

The initial and final permutations consume a 64-bit sequence and produce a sequence of bits according to a specific permutation pattern. The pattern for the initial permutation is illustrated in Figure 4. We implemented the permutations using a lookup table as shown in Figure 3. The `permute` method loops through the output bit

```
public value class Unsigned<T extends Enum<T>> {
    ...
    Unsigned<T> permute ( (value T[T]) permTable ) {
        bit newBits[T];
        for ( T i ) {
            newBits[i] = data[permTable[i]];
        }
        return new Unsigned<T>(newBits);
    }
    ...
}
```

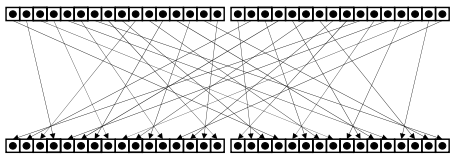

Fig. 4. Permutation pattern for `IP`.

```
// initial permutation (IP)
import static DES.sixtyfour.*;

public value class IP {
    public static (value sixtyfour[sixtyfour]) Permutation = {
        b57, b49, b41, b33, b25, b17, b9,  b1, b59, b51, b43, b35, b27, b19, b11, b3,
        b61, b53, b45, b37, b29, b21, b13, b5, b63, b55, b47, b39, b31, b23, b15, b7,
        b56, b48, b40, b32, b24, b16, b8,  b0, b58, b50, b42, b34, b26, b18, b10, b2,
        b60, b52, b44, b36, b28, b20, b12, b4, b62, b54, b46, b38, b30, b22, b14, b6
    };
    ...
}
```

Fig. 3. DES code snippets showing initial permutation.

indexes in order, and maps the appropriate input bit to the corresponding output bit. The enumerations and their iterators make it possible to readily name each individual bit, and as a result, bit-permutations are easy to implement. The ability to specify transformations at the bit-level provides several advantages for hardware synthesis. Namely, the explicit enumeration of the bits decouples their naming from a platform-specific implementation, and as a result there are no bit-masks or other bit-extraction routines that muddle the code. Furthermore, the enumeration of the individual bits means we can closely match permutations and similar transformations to their Verilog or VHDL counterparts. As a result, the compiler can command a lot of freedom in transforming the code. It has also been shown that such a bit-level representation of the computation leads to efficient code generation for conventional architectures and processors that support short-vector instructions [5, 6]. There are also various benefits for a programmer. For example, the `permute` method can process the input or output bits in any order, according to what is most convenient. Similarly, off-by-one errors are avoided, through the use of `enum`-indexed arrays.

The `Fiestel` method performs the transformations illustrated in Figure 6. The 32-bit `R` half block undergoes an expansion to 48-bits, and the result is mixed with an encryption key using an `xor` operation. The result is then split into eight 6-bit pieces, each of which is substituted with a 4-bit value using a unique substitution box (`Substitutes[i]`). The eight 4-bit resultant values are concatenated to form a 32-bit half block that is in turn permuted. The `Fiestel` method and coding rounds run in hardware on the FPGA. The `main` method, shown below, runs in software on the CPU.

```
public static void main(String[] argv) {
    Unsigned64 key  = makeUnsigned64("0xFEDCBA9876543210");
    Unsigned64 text = makeUnsigned64("0x0123456789ABCDEF");

    KeySchedule keys   = new KeySchedule(key);
    Unsigned64  cipher = DEScoder(keys, text);
    System.out.println(Long.toHexString(cipher.longValue()));
}
```

```
static Unsigned32 Fiestel(KeySchedule keys, Sixteen round, Unsigned32 R) {
    // half-block expansion
    Unsigned48 E = expand(R);

    // key mixing
    Unsigned48 K = keys.keySchedule(round);
    Unsigned48 S = E ^ K;

    // substitutions
    Unsigned4 Substitutes[eight];

    fourtyeight startBit = fourtyeight.b0;
    for ( eight i ) {
        // extract 6-bit piece
        fourtyeight endBit = startBit + fourtyeight.b5;
        Unsigned6 bits = S.extractSixBits(startBit, endBit);

        // substitute bits
        Substitutes[i] = Sbox(i, bits);

        // move on to next 6-bit piece
        startBit += fourtyeight.b6;
    }

    // concatenate pieces to form
    // a 32-bit half block again
    thirtytwo k;
    bit[thirtytwo] pBits;
    for ( eight i ) {
        for (four j) {
            pBits[k] = Substitutes[i].data[j];
        }
    }

    // permute result and return
    Unsigned32 P = new Unsigned32(pBits);
    return reversePermute(P);
}
```

Fig. 5. DES Fiestel round.

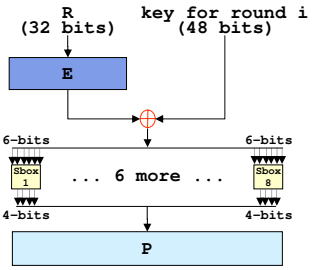

Fig. 6. Block level diagram of Fiestel round.

The program exercises co-execution between hardware and software, and demonstrates the use of varying object sizes and object-oriented features in hardware.

4 From Lime to the Virtual Machine

Lime programs can be compiled to regular Java bytecode and executed on any Java VM. The Lime bytecode generation performs two additional steps than the Java compiler. First, the Lime compiler generates bytecode to add "value" to value types:

- Default constructors, `equals(Object o)` and `hashCode()` methods are created for those value classes that do not define them.
- Uninitialized variables of value types are rewritten with default initializers.
- Operator definitions listed in Section 2.1 are added for value `enum`'s. Value types that support `++`, `--`, `<` and `>` operators have the range operator, `::`, defined for them.
- Operator expressions are converted to appropriate operator method calls. E.g., `x == y` is converted to `x.equals(y)`, assuming `x` is of value type.

For the purpose of separate compilation, all value types are translated to implement the `lime.lang.Value` interface. When loaded as a binary class, this interface indicates

to the Lime compiler that it is a value class. Additional interfaces are added for value types supporting different operators. For example, all value types supporting operator `<` implement the interface `lime.lang.HasGT<T>`, where `HasGT<T>` contains one operator, `boolean this < (T op2)`.

Next, instantiations of value generic types must be expanded. Generics is a powerful abstraction tool for programmers. However, generic value classes also significantly complicate our compilation process. To see why, consider generating a default constructor for `Unsigned<W>`. This constructor needs to initialize the `data` field to a `bit`-array of size w, where w is the number of values defined for `enum` type `W`. However, the value of w changes for each concrete instantiation of `W`. We have no way of initializing this field without knowing what `W` is type-instantiated to. For this reason, the erasure-based compilation technique used by Java generics is not applicable. We must employ an expansion-based compilation scheme, where each instantiation of `Unsigned<W>` creates a different type.

Java generic classes not annotated with the `value` modifier are translated using the standard erasure technique, as long as they do not instantiate generic value types with their type variables. As a result, pure Java code that is compiled with our compiler remains backward compatible.

There are of course numerous optimizations that exploit bit-width information and the immutable properties of value types (see Section 2.3 for examples). Such optimizations are well studied and understood. In this paper, we primarily focus on the less understood parts of our language, such as translating Object-Oriented semantics down to the hardware fabric.

The Lime frontend compiler (source to bytecode or spatial intermediate representation) is implemented using the JastAdd extensible Java compiler [7].

5 Liquid Metal Runtime for Mixed-mode Execution

A Lime program may run in mixed-mode. That is, some parts of the program will run in the virtual machine, and some parts will run in hardware (FPGA). An example mixed-mode architecture is a CPU coupled with an FPGA coprocessor, or a desktop workstation with an FPGA PCI card. Yet another example is an FPGA with processors that are embedded within the fabric. We use a Xilinx Virtex-4 board as an instance of the latter. The Virtex-4 is also our evaluation platform for this paper. Programs that run in software use its embedded IBM PowerPC 405 which runs at a frequency of 300 MHz. The processor boots an embedded Linux kernel and can run a JVM.

The Liquid Metal runtime (LMRT) provides an API and a library implementation that allows a program to orchestrate its execution on a given computational platform. It simplifies the exchange of code and data between processing elements (e.g., PowerPC and FPGA), and automatically manages data transfer and synchronization where appropriate. The API calls are typically generated automatically by our compiler, although a programmer can make use of the API directly and manually manage the computation when it is desirable to do so.

The LMRT organizes computation as a set of code objects and buffer objects. A buffer is either an input buffer, an output buffer, or a shared buffer. A code object reads input data from an attached input buffer. Similarly it writes its output to an attached output buffer. Data is explicitly transferred (copied) between input and output buffers. In contrast, a shared buffer simultaneously serves as an input and output buffer for multiple code objects. All communication between code objects is done through buffers.

5.1 Code objects

The LMRT assumes there is a master processing element that initiates all computation. For example, the VM running on the PowerPC processor serves as the master on our Virtex board. The VM can invoke the LMRT API through JNI. The master creates code objects, attaches input and output buffers, and then runs, pauses, or deletes the code object as the computation evolves.

A code object embodies a set of methods that carry out computation. It can contain private mutable data that persists throughout its execution (i.e., stateful computation). However, code objects are not allowed to maintain references to state that is mutated in another object.

A Lime program running wholly in the virtual machine can be viewed as a code object with no input or output buffers. A program running in mixed-mode consists of at least two code objects: one running in software, and the other running in hardware. Data is exchanged between them using buffer objects.

5.2 Buffer objects

A buffer is attached to a code object which can then access the buffered data using read and write operators. The LMRT defines three modes to read data from or write data to a buffer.

- **FIFO:** The buffer is a first-in first-out queue, and it is accessed using push or pop methods. For example, code running in the VM can push data into the buffer, and code running in the FPGA pops data from the buffer.
- **DMA:** The buffer serves as a local store, with `put` operations to write data to the buffer, and `get` operations to read data from it. The put and get commands operate on contiguous chunks of data.
- **RF:** The buffer serves as a scalar register file, shared between code objects.

The LMRT makes it possible to decouple the application-level communication model from the implementation in the architecture. That is, a buffer decouples (1) the program view of how data is shared and communicated between code objects from (2) the actual implementation of the I/O network in the target architecture. Hence a program can use a pattern of communication that is suitable for the application it encodes, while the compiler and runtime system can determine the best method for supporting the application-level communication model on the architecture.

5.3 The LMRT Hardware Interface Layer

One of the main reasons for the LMRT is to automatically manage communication and synchronization between processing elements. In a mixed-mode environment, communication between the VM and FPGA has to be realized over a physical network interconnecting the FPGA with the processor where the VM is running.

In our current Virtex platform, we use the register file (RF) interface between the processor and the FPGA. The RF is synthesized into the fabric itself. It is directly accessible from the FPGA. From the processor side, the registers are memory mapped to a designated region of memory. The RF we use consists of 32 registers, each 32 bits wide. The 32 registers are portioned into two sets. The first is read accessible from the FPGA, but not write accessible. Those registers are read/write accessible from the VM.

The second set is read accessible from the VM, but not write accessible. The registers in the second set are read/write accessible from the FPGA.

The FIFO and DMA communication styles are implemented using the RF model. The FIFO model maintains head and tail pointers and writes the registers in order. The DMA model allows for 15x32 bits of data transfer, with 32 bits used for tags. While we use a register file interface between the VM and the FPGA, other implementations are feasible. Namely, we can implement a FIFO or a DMA directly in the FPGA fabric, and compile the code objects to use these interfaces. This kind of flexibility makes it possible to both experiment with different communication models, and adapt the interconnect according to the characteristics of the computation.

6 From Lime to a Spatial Intermediate Representation

Compiling a Lime program to execute on the FPGA requires a few transformations. Some transformations are necessary to correctly and adequately handle object orientation in hardware. Others are necessary for exposing parallelism and generating efficient circuits. Performance efficiency in the FPGA is attributed to several factors [8]:

1. **Custom datapaths:** a custom datapath elides extraneous resources to provide a distinct advantage over a predefined datapath in a conventional processor.
2. **Multi-granular operations:** a bit-width cognizant datapath, ALUs, and operators tailor the circuitry to the application, often leading to power and performance advantages.
3. **Spatial parallelism:** FPGAs offer flexible parallel structures to match the parallelism in an application. Hence bit, instruction, data, and task-level parallelism are all plausible forms of parallelism. We refer to parallelism in the FPGA as spatial since computation typically propagates throughout the fabric.

In this paper we focus exclusively on the issues related to discovering spatial parallelism and realizing such parallelism in hardware. Toward this purpose, we employ a spatial intermediate representation (SIR) that facilates the analysis of Lime programs. The SIR also provides a uniform framework for refining the inherent parallelism in the application to that it is best suited for the target platform.

6.1 Spatial Intermediate Representation

The SIR exposes both computation and communication. It is based on the synchronous dataflow model of computation [9, 10]. The SIR is a graph of *filters* interconnected with communication channels. A filter consists of a single *work* method that corresponds to a specific method call derived from a Lime program. A filter may contain other methods that are called *helpers*. The difference between the work method and the helpers is that only the work method may read data from its input channel or write data to its output channel.

For example, each static call to `permute()` in the DES example corresponds to a specific filter in the SIR. A filter consumes data from its input channel, executes the work method, and writes its results to an output channel. The input and output of the `permute` method that performs the initial permutation is an `Unsigned64` value. Hence, the work method for `permute` consumes 64 bits and produces 64 bits on every execution. The filter work method runs repeatedly as long as a sufficient quantity of

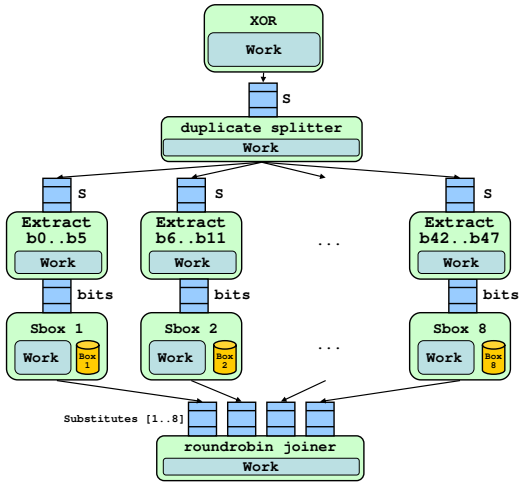

Fig. 7. SIR example for box substitutions in DES.

input data is available. Filters are independent of each other, do not share state, and can run autonomously.

Filters have a single input channel and a single output channel. A filter may communicate its output data to multiple filters by routing the data through a *splitter*. A splitter can either duplicate the input it receives and pass it on to its siblings, or it can distribute data in a roundrobin manner according to a specified set of weights. The splitter's counterpart is a *joiner*. A joiner collects and aggregates data from multiple filters in a roundrobin manner, and routes the resultant data to another filter. The single-input to single-output restriction placed on filters, and the routing of data through splitters and joiners for fan-out and fan-in imposes structure on the SIR graphs. The structure can occasionally lead to additional communication compared to an unstructured graph. In DES, this occurs between `Fiestel` rounds where the values of `L` and `R` are interchanged[4]. However we believe that the benefits of a structured SIR outweigh its drawbacks, and prior work has shown that structured graphs can be practically refined to their unstructured counterparts [11].

The SIR graph in Figure 7 illustrates an example derived from the box substitutions (`Sbox`) that occur in the `Fiestel` rounds. In the Figure, the output of the `xor` operator is duplicated to eight filters labeled `Extract`, each of which implements the `extractSixBits` methods but for different bit indexes. For example, the left-most filter labeled `Extract b0..b5` inputs a 32-bit value and always extracts a value consisting of the bits at locations `b0..b5`. Similarly, the `Extract b42..b47` filter always extracts the bits `b42..b47`. The output of the former is the input to the `Sbox 1` filter which performs the appropriate bit substitutions for bits `b0..b5`. The `Extract` and `Sbox` filters make up a producer-consumer pair and are said to form a *pipeline*. Pipelines in the SIR graph expose pipeline parallelism that is readily exploited in hardware. The output of each `Sbox` is routed to a joiner that collects each of the 4-bit pieces in a roundrobin manner and outputs a 32-bit half block.

[4] Figure 2 illustrates unstructured communication. It is left as an exercise for the reader to determine the structured SIR equivalent.

Filters, like objects, may have fields. The fields are initialized using an *init* method whose parameters must be resolved when the SIR is constructed. Each of the `Extract` filters is initialized with the start and end bits that it is responsible for. Similarly, each of the `Sbox` filters is initialized with a table that encodes the unique substitution pattern for the bits it is responsible for. The fields of a filter cannot be shared and are conceptually stored in a local memory that is exclusive to that filter. In the Figure 7, the cylinders labeled `Box 1..8` store the substitution boxes. The `Extract` method requires no storage since the initialization parameters are constant-propagated throughout the filter work method.

6.2 Compiling Lime to SIR

There are three key considerations in translating a Lime program into a spatial representation. We must determine the dataflow of the program: which objects (or primitive values) need to be passed from filter to filter, and which can be statically initialized (or calculated from statically initialized variables). We must also determine what constitutes a filter: what Lime code is a filter responsible for executing? Lastly, we must determine how important object-oriented features can be supported in hardware: how are objects represented? How do we support virtual method dispatch? How do we handle object allocation?

Answering these questions requires us to first construct a control flow graph from program entry to exit, including inlining recursive method calls[5]. The only cycles the control flow graph can have are those produced by Lime's looping primitives, such as `for` or `while`. The inlining of recursive method calls necessarily places a restriction on the type of programs that can be synthesized into hardware: programs involving recursive method calls that are not statically bounded are out of the reach of synthesis. The basic approach is to construct a dataflow graph of non-static data in a program. Methods that receive non-static data as input are conceptually mapped to filters. The flow of data between methods outlines the overall SIR topology.

Determining Dataflow. We use constant propagation to determine which variables have statically computable values. For example, in `for (eight i) { ... }` used in the box substitution in `Fiestel`, the variable `i` is statically initialized to be `eight.b0`, and subsequently updated by `i + eight.b1` during each iteration. This updated value can be computed from statically known values. Thus, `i` does not need to be an input to a filter work method. Instead, it is used as a filter initializer or mapped to a filter field. On the other hand, `bits` is initialized by expression `S.extractSixBits(startBit, endBit)`. `S` does not have a statically computable value—its value depends on the filter input to method `Fiestel`. Thus, the computation of `S.extractSixBits(startBit, endBit)` requires `S` as an input. (Note that the receiver of a method invocation is considered an input, as well.) Consequently, `bits` is not statically computable either, and must be the output of the filter/pipeline for the expression `S.extractSixBits(startBit, endBit)`. Using standard dataflow techniques, we can determine the data necessary at each point of the program.

Defining Filters. The identifying characteristic of filters is that they perform input or output (I/O) of data that is not statically computable. Once we determine what data

[5] There is no good way to deal with unbounded recursion in hardware.

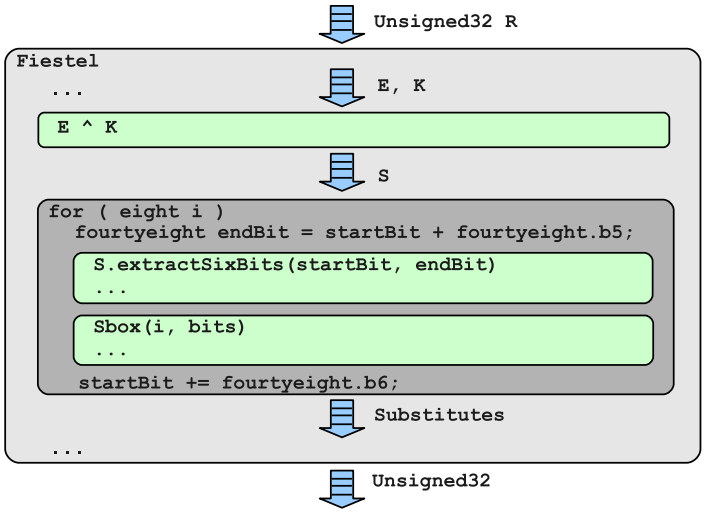

Fig. 8. I/O containers for Fiestel.

is needed for input and output at each program location, we decompose the program into (possibly nested) I/O "containers", and then construct filters and pipelines from these containers.

The entry and exit of a Lime method form natural bounds for an outermost I/O container. For example, an outermost I/O container is constructed for method `Fiestel`. Within these bounds, we identify two types of I/O containers.

First, an I/O container is indicated by a method or constructor invocation, where at least one of the arguments (including `this`, if method call is not static) has been identified as a filter input. For example, `S.extractSixBits(startBit, endBit)` in `Fiestel` becomes an I/O container, with `S` as its input. We then analyze the declaration of `extractSixBits`, and inline the I/O containers for the method declaration inside the container for the method invocation.

A second type of I/O container is formed from branching statements such as `for` loops, or `if/else`, where the body of a branch references filter inputs. Each branching container may include nested containers depending on the body of the branch. For example, the box substitution `for (eight i)` { ... } loop in `Fiestel` becomes a branching I/O container. Nested within it, are a series of containers, such as the one for method call `S.extractSixBits(startBit, endBit)`, as well as a container of `Sbox(i, bits)`.

Figure 8 illustrates the I/O containers identified for `Fiestel` in Figure 3. Note that expression `E^K` constitutes an I/O container because operator `^` is defined for `Unsigned`. Thus, `E^K` is turned into method invocation `E.$XOR(K)`. Also note that non-I/O statements, such as loop index update (e.g., `sIndex += fourtyeight.b6;`), become local to their enclosing I/O container. For space reasons, `...` represents elided I/O containers.

SIR from I/O containers. An I/O container has a natural mapping down to the SIR. An I/O container with no nested containers naturally maps to a filter. Its work method contains all the statements enclosed by the container. These are generally arithmetic computations that have a straight-forward mapping to hardware. If

these statements involve any static references, the definitions of the referenced data or methods are declared as local fields in the filter or as local variables in the work method.

Filters (or pipelines) from I/O containers at the same nesting level are connected to form a pipeline. Thus, an I/O container with nested containers is mapped down to a pipeline formed by its children.

A branching I/O container that is formed by a `for` statement, creates a more interesting mapping to the SIR. First, the work statements or nested I/O containers within the loop body are turned into a filter (or pipeline, respectively). If the loop iterations are independent of each other with respect to the filter input and output data, then the filter (pipeline) that makes up the loop body is considered data-parallel. It can be replicated once for each iteration of the loop. This basically translates the Lime code to a data-parallel representation in the SIR. A data splitter is added at the beginning of the `for` I/O container. The splitter *duplicates* the incoming data, and sends it down each replicated loop body filter (pipeline). Data that is not part of the filter input and that may depend on the loop index are used as *init* values for the filter construction. A joiner is then added at the exit of the `for` I/O container to assemble the output of each replicated filter (pipeline).

When we cannot determine that the loop iterations are independent, we have to explore an alternative mapping. In the case, the computation is considered stateful. In this case, we can statically unroll the loop and connect the unrolled loop body filter (pipeline) sequential to form longer pipelines. Alternatively, we can create a feedback loop such that the output of the loop body filter (pipeline) feeds back into itself. This second option, however, is untested in our SIR compiler.

Similar split/join structures are generated for other branching statement I/O containers. Applying these rules, it is easy to see how I/O containers from Figure 8 can be mapped to exactly the SIR structure in Figure 7.

Object representation in Hardware. The most general way an object can be represented in hardware is by serializing it into an array of bits that is either packed into registers, or stored in memory. The kind of Lime programs most amenable for synthesis to hardware use data with efficient representations. Lime's language design is geared toward exposing such representations from a high level, as we illustrated in Section 2. Objects of value types have no mutable state, and thus can be safely packed into registers, instead of being stored in much slower memory.

Dynamic dispatch in hardware. One of the defining features of object-oriented paradigms is the dynamic dispatch of methods. In order to perform dynamic dispatch in hardware, we assign a unique identifier to each type, which is then carried by the object of that type. Thus, object representation may require bits for the type identifier to be serialized, as well. When mapping an I/O container resulting from a virtual method invocation to SIR filters, we must generate a pipeline for each possible target method of the virtual call. All pipelines from target methods are then added to a *switch* connector. The condition for the switch is the type identifier that is carried by the incoming `this` object. A pipeline of the target method is only invoked if the type identifier of the input `this` object is equal to the type identifier of the method's declaring class, or one of its subclasses. We use analysis such as RTA [12] to reduce the number of potential target methods that need to be synthesized. If the target method

of a virtual call can be statically identified, then the object does not have to carry a type identifier.

Object allocation in hardware. Lime programs can use the `new` keyword to create new objects. However, laying out a program in hardware means all memory needed must be known ahead of time. Thus, a program for synthesis must be able to resolve statically all `new`'s, and space is allocated in registers or memory. Repeatedly `new`-ing objects in an unbounded loop, with the objects having lifetimes persisting beyond the life of the loop, is not permitted in synthesized programs.

7 SIR Compiler

The SIR that we adopt is both a good match for synthesis and also convenient for performing coarse-grained optimizations that impact the realized parallelism. We build heavily on the StreamIt compiler [13] to implement our SIR and our SIR compiler. The StreamIt compiler is designed for the StreamIt programming language. In StreamIt, programs describe SIR graphs algorithmically and programmatically using language constructs for filters, pipelines, splitters/joiners, and feedback loops. The latter create cycles in the SIR graph although we do not currently handle cycles.

7.1 Lowering Communication

The SIR compiler transforms the SIR graph to reduce the communication overhead and cost. In an FPGA, excessive fan-out and fan-in is not desirable. Hence the compiler attempts to collapse subgraphs that are dominated by a splitter and post-dominated by a joiner. This transformation is feasible when the filters that make up the subgraph are stateless. In a Lime program, methods of a value class are stateless. For example, the `Extract` and `Sbox` filters in the SIR graph shown in Figure 7 are stateless since neither of the two has any mutable state. However, since each of these filters is specialized for a specific set of bits, collapsing the subgraph results in at least one stateful filter, namely the `Sbox` filter in this case. The collapsed graph is shown in Figure 9. Each execution of the work method updates the state of the filter (shown as `i` in the Figure) so that on the first execution it performs the substitution that correlates with `Sbox 8`, on its second execution it performs the substitution for `Sbox 7`, and so on until its ninth execution where it resets the state and resumes with `Sbox 8`.

The `Extract` filter does not need to keep track of its execution counts if the compiler can determine that each of the `Extract` filters in the original graph operated in order on mutually exclusive bits. Such an analysis requires dataflow analysis within the filter work method, and is aided by very aggressive constant propagation, loop unrolling, and dead code elimination. More powerful analysis is also possible when filters carry out affine computations [14, 15, 6]. The SIR compiler employs these techniques to reduce overall communication. The impact on the generated hardware can be significant in terms of speed (time) and overall area (space). We demonstrate the space-time tradeoff by synthesizing the SIR graphs in Figures 7 and 9. The results for these two implementations appear as `Sbox Parallel Duplicate` and `Sbox State` respectively in Figure 10. The evaluation platform is a Virtex-4 FPGA with an embedded PowerPC processor (PPC). The speedup results compare the performance of

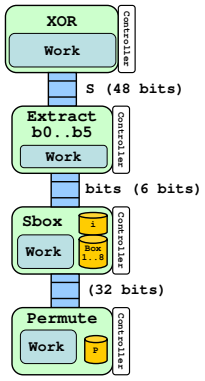

Fig. 9. Result of collapsing SIR shown in Figure 7.

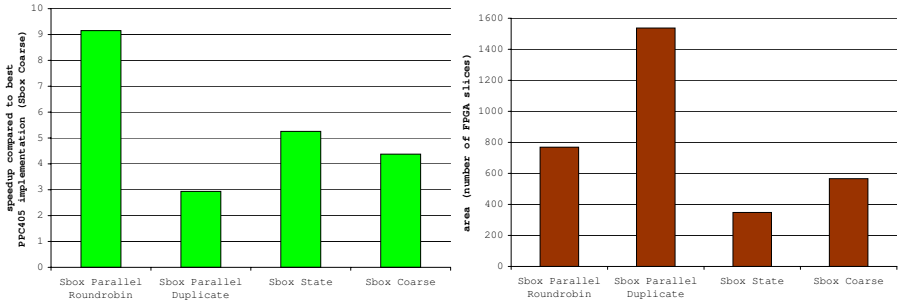

Fig. 10. Speedup and area results for different SIR realizations of `Sbox`.

each hardware implementation to the implementation that yields the best results on the PPC.

We also performed two other transformations: `Sbox Parallel Roundrobin` and `Sbox Coarse`. The former uses dataflow analysis to determine that a roundrobin splitter can replace the duplicate splitter and the `Extract` filters in Figure 7. The latter eliminates the state from the `Sbox` filter in Figure 9 by substituting all 48 bits in one execution of the work method. The results are as one should expect. The fastest hardware implementation uses the roundrobin splitter and parallel `Sbox` filters. This implementation is roughly 3x faster than the duplicate splitter implementation and 100% more space efficient since the roundrobin splitter avoids needless communication and optimizes the datapaths between filters aggressively. The area overhead is 50% larger than that of the most compact implementation, namely `Sbox State` which is a pipelined implementation with an aggressively optimized datapath. The coarse implementation is the slowest of the four variants since it performs the most amount of work per execution of the work method and affords little opportunity for pipeline parallelism. It is however the best implementation for software although it is worthy to note that it does not use the natural width of the machine in this case. In other words, the version of `Sbox Coarse` that we benchmark in software uses the same granularity as the FPGA and runs the work methods at bit-granularity. This purpose of the performance comparison is to illustrate the space-time trade-off that exists. In Section 9 we compare our synthesis results to various optimized baselines.

7.2 Load Balancing

The SIR compiler also attempts to refine the SIR graph to realize a more load-balanced graph. This is important because it minimizes the effects of a bottleneck in the overall design. Toward this end, we currently use the heuristics and optimizations described in [11] and implemented in the StreamIt compiler. The compiler uses various heuristics to fuse adjacent filters when it is profitable to do so. The heuristics rely on a work-estimation methodology to detect load imbalance. In our case, work-estimation is a simple scalar measure of the critical path length through the filter work method. It is calculated using predefined latencies for individual primitive operations. We believe however that there are other ways of dealing with load imbalance on an FPGA platform but we have not yet thoroughly investigated alternatives.

8 Getting to Hardware

The last step in our toolflow is HDL code generation. It is accomplished using our SIR to Verilog compiler called Crucible. It compiles each filter in the SIR to a Verilog hardware module. It then assembles the modules together according to the dataflow edges in the SIR graph. The Crucible also generates the HDL interfaces used to exchange data between the processor and the FPGA in order to support mixed-mode execution. The interfaces work in conjunctions with the Liquid Metal runtime to provide the network between processing elements, as well as the API implementation from the FPGA side. The completed design is finally synthesized using commercial synthesis tools to produce a bitstream that can be used to program the FPGA. We use the Xilinx synthesis tool (XST) for this purpose. FPGAs typically require vendor specific tools, so for other targets, the appropriate synthesis tool is used. The Crucible controls and guides the synthesis tool by setting appropriate synthesis parameters that impact resource allocation policies, arithmetic circuit implementation, the placement of objects in FPGA memory, etc. The Crucible is best suited to guide these policies since it has a global view of the application.

The Crucible address both micro-functional (intra-filter) and macro-functional (inter-filter) synthesis issues. It extends the Trimaran [16] compiler with optimizations and heuristics that are space-time aware. We leverage many of the existing analysis and optimizations in Trimaran to optimize the code within each filter. These optimizations include critical path reduction, region formation for instruction-level parallelism, predication, vectorization, and aggressive instruction scheduling algorithms. In addition, the Crucible is bit-width cognizant, and although the compiler can perform bit-width analysis, we primarily rely on the high level semantics of the Lime program to elide or augment the analysis where feasible.

In the micro-functional sense, the compiler operates on a control flow graph (CFG) consisting of operations and edges. Operations are grouped into basic blocks. Basic blocks are in turn grouped into procedures. Each procedure typically represents a filter. The code generation applies a bottom-up algorithm to the CFG, starting with the operations. It generates Verilog for each operation, then routes the operands between them. Basic blocks serve as a hierarchical building block. They are composed together with dataflow edges, eventually encompassing the entire procedure. Since procedures represent filters, it is also necessary to generate the FIFOs that interconnect them according to the SIR. The size of each FIFO is either determined from the SIR according to the data types exchanged between filters, or using a heuristic that is subject to

Table 1. Comparison of DES implementation on different processing platforms.

processor	PPC 405	FPGA	Pentium-II	Core 2 Duo
frequency	300 MHz	129 MHz	400 MHz	2.66 GHz
throughput	27 Mbps	30 Mbps	45 Mbps	426 Mbps
performance	1	1.11	1.69	16
DES version	C reference	Lime	C reference	C reference

space-time constraints. This is an example of a macro-functional optimization. If too little buffering is provided, then throughput decreases as modules stall to send or receive data; whereas too much buffering incurs substantial space overheads. Macro-functional optimizations require careful consideration of area and performance trade-offs to judiciously maximize application throughput at the lowest costs.

In addition to the buffering considerations, the Crucible also generates hardware controllers that stage the execution of the filters in hardware. The results presented in this paper use a basic execution model that executes the filter work methods when the input data is ready, and reads from an empty channel (writes to full channel) block the filter under the channel until other filters make progress.

A greater description of the Crucible and its optimizations are beyond the scope of this paper.

9 Experimental Results

We compiled and synthesized the DES Lime code from Section 3 to run in an FPGA. We measured the application throughput at steady state in terms of Mbits per second second (Mbps). We compare our results to an optimized implementation of DES (reference implementation) running on an Intel Pentium-II at 400 MHz, a Core 2 Duo processor with a frequency of 2.66 GHz, and a 300 MHz PPC 405 which is the embedded processor available in the Virtex-4 LX200. The frequency of the DES design generated from the Lime program is 129 MHz. The results are summarized in Table 1. The row labeled **performance** shows the relative throughput compared to the PPC 405. The PPC is a reasonable baseline since it is manufactured in the same technology as the FPGA fabric. Compared to the embedded processor, the FPGA implementation is 11% faster. It is 66% slower than a reasonably optimized DES coder running on a Pentium-II, and 14x slower than the fastest processor we tested.

The results show that we can achieve a reasonable hardware implementation of DES starting from a high level program that was relatively easy to implement. Compared to reference C implementations that we found and studied, we believe the Lime program is easier to understand. In addition, the Lime program is arguably more portable since computation is explicitly expressed at the bit-level and is therefore platform independent. This is in contrast to software implementations that have to match the natural processing width of their target platforms and hence express computation at the granularity of bytes or words instead of bits. We believe that starting with a bit-level implementation is more natural for a programmer since it closely follows the specification of the algorithm.

The FPGA implementation that we realized from the Lime program requires nearly 84% of the total FPGA area. This is a significant portion of the FPGA. The area requirement is high because we are mapping the entire DES coder pipeline (all 16 rounds)

to hardware and we are not reusing any resources. The spatial mapping is straightforward to realize but there are alternative mapping strategies that can significantly reduce the area. Namely, sharing resources and trading off space for throughput is an important consideration. We showed an example of this kind of trade-off earlier using the `Sbox` code (refer to Figure 10). We believe that there is significant room for improvement in this regard and this is an active area of research that we are pursuing.

Our goal however is not to be the world's best high-level synthesis compiler. Rather, our emphasis is on extending the set of object-oriented programming features that we can readily and efficiently implement in hardware so that skilled Java programmers can transparently tap the advantages of FPGAs. In the current work, we showed that we can support several important features including value types, generics, object allocation, and operator overloading. We are also capable of supporting dynamic dispatch in hardware although the DES example did not provide a natural way to showcase this feature.

10 Related Work

10.1 Languages with Value Types

Kava [1] is an early implementation of value types as lightweight objects in Java. The design of Lime is very much inspired by Kava. However, Kava was designed before `enum` types or generics were introduced into Java. Thus, Kava chose a different type hierarchy which put value types and `Object` at the same level. This design does not fit in well with the current Java design. Lime remedied this by using a `value` modifier. Lime also provides support for value generic types. Additionally, Kava value types are not automatically initialized, nor are default constructors generated.

C# [17] offers value types in the form of structs. One important difference between C# value types and Lime value types is that C# value types cannot inherit from other value types. Inheritance and dynamic dispatch of methods are key features of the OO paradigm. Value types should be able to take advantage of these abstractions. Furthermore, C# struct references must be manually initialized by the programmer, even though a default constructor is provided for each struct. Lime value type references are automatically initialized, similar to the way primitive types are treated.

Recent work by Zibin et al. [18] has shown a way to enforce immutability using an extra immutability type parameter. In this work, a class can be defined such that it can be used in a mutable or immutable context. In Lime, a value class and a mutable class must be separately defined. The method proposed in [18] is an interesting way to integrate a functional style with Java's inherently mutable core. We could incorporate similar techniques in Lime in the future.

10.2 Synthesizing High-Level Languages

Many researchers have worked on compilers and new high-level languages for generating hardware in the past few years. Languages such as SystemC [4] have been proposed to provide the same functionality as lower-level languages such as Verilog and VHDL at a higher-level of abstraction. SystemC is a set of library routines and macros implemented in C++, which makes it possible to simulate concurrent processes, each described by ordinary C++ syntax. Similarly, Handle-C [19] is another hardware/software construction language with C syntax that support behavioral description of hardware. SA-C

[20] is a single assignment high-level synthesizable language. An SA-C program can be viewed as a graph where nodes correspond to operators, and edges to data paths. Dataflow graphs are ideal (data driven, timeless) abstractions for hardware circuits.

StreamC [21] is a compiler which focuses on extensions to C that facilitate expressing communication between parallel processes. Spark [22] is another C to VHDL compiler which supports transformations such as loop unrolling, common sub-expression elimination, copy propagation, etc. DEFACTO[23] and ROCCC[24] are two other hardware generation systems that take C as input and generate VHDL code as output. To the best of our knowledge, none of these compilation systems support high level object-oriented techniques.

Work by Chu[25] proposes object oriented circuit-generators. Circuit-generators, parameterized code which produces a digital design, enable designers to conveniently specify reusable designs in a familiar programming environment. Although object oriented techniques can be used to design these generator, this system is not intended for both hardware and software programming in a parallel system. Additionally, the syntax used in the proposed system is not appropriate for large-scale object oriented software designs.

11 Conclusion

In this paper, we introduce Lime, an OO language for programming heterogeneous computing environments. The entire Lime architecture provides end-to-end support from a high-level OO programming language, to compilation to both the Java VM, the FPGA, and a runtime that allows mixed-mode operation such that code can run on partly on the VM and partly on the FPGA, delegating work to the most optimal fabric for a certain task. Lime is a first step toward a system that can "JIT the hardware", truly taking advantage of the multitude of computing architectures.

Acknowledgments

This work is supported in part by IBM Research and the National Science Foundation Graduate Research Fellowship. We thank Bill Thies and Michael Gordon of MIT for their help with the StreamIt compiler, Stephen Neuendorffer of Xilinx for his help with the Xilinx tools and platforms, and the reviewers for their helpful comments and appreciation of our vision.

References

1. Bacon, D.F.: Kava: A Java dialect with a uniform object model for lightweight classes. Concurrency—Practice and Experience **15** (2003) 185–206
2. Wu, P., Midkiff, S.P., Moreira, J.E., Gupta, M.: Efficient support for complex numbers in java. In: Java Grande. (1999) 109–118
3. IEEE: 1076 IEEE standard VHDL language reference manual. Technical report (2002)
4. IEEE: IEEE standard SystemC language reference manual. Technical report (2006)
5. Narayanan, M., Yelick, K.A.: Generating permutation instructions from a high-level description. In: Workshop on Media and Streaming Processors. (2004)

6. Solar-Lezama, A., Rabbah, R., Bodík, R., Ebcioğlu, K.: Programming by sketching for bit-streaming programs. In: PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, New York, NY, USA, ACM (2005) 281–294
7. Ekman, T., Hedin, G.: The jastadd extensible java compiler. In: OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object oriented programming systems and applications, New York, NY, USA, ACM (2007) 1–18
8. Babb, J., Frank, M., Lee, V., Waingold, E., Barua, R., Taylor, M., Kim, J., Devabhaktuni, S., Agarwal, A.: The raw benchmark suite: Computation structures for general purpose computing. In: Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines. (1997)
9. Lee, E.A., Messerschmitt, D.G.: Static scheduling of synchronous data flow programs for digital signal processing. IEEE Trans. on Computers (1987)
10. Bhattacharyya, S.S., Murthy, P.K., Lee, E.A.: Software Synthesis from Dataflow Graphs. Kluwer Academic Publishers (1996)
11. Gordon, M., Thies, W., Amarasinghe, S.: Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems. (2006)
12. Bacon, D.F.: Fast and effective optimization of statically typed object-oriented languages. PhD thesis (1997)
13. StreamIt: http://cag.csail.mit.edu/streamit (2003)
14. Lamb, A.A., Thies, W., Amarasinghe, S.: Linear Analysis and Optimization of Stream Programs. In: PLDI. (2003)
15. Agrawal, S., Thies, W., Amarasinghe, S.: Optimizing stream programs using linear state space analysis. In: CASES. (2005)
16. Trimaran Research Infrastructure: http://www.trimaran.org (1999)
17. Hejlsberg, A., Wiltamuth, S., Golde, P.: C# Language Specification. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2003)
18. Zibin, Y., Potanin, A., Ali, M., Artzi, S., Kieżun, A., Ernst, M.D.: Object and reference immutability using Java generics. In: ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Dubrovnik, Croatia (2007)
19. Handle-C Language Overview: http://www.celoxica.com (2004)
20. Najjar, W., Bohm, W., Draper, B., Hammes, J., Rinker, R., Beveridge, J., Chawathe, M., Ross, C.: High-level language abstraction for reconfigurable computing (2003)
21. Mencer, O., Hubert, H., Morf, M., Flynn, M.J.: Stream: Object-oriented programming of stream architectures using PAM-blox. In: FPL. (2000) 595–604
22. Gupta, S.: Spark: A high-level synthesis framework for applying parallelizing compiler transformations (2003)
23. Diniz, P.C., Hall, M.W., Park, J., So, B., , Ziegler, H.E.: Bridging the gap between compilation and synthesis in the defacto system. In: Lecture Notes in Computer Science. (2001) 52–70
24. Guo, Z., Buyukkurt, B., Najjar, W., Vissers, K.: Optimized generation of data-path from c codes for fpgas. In: Design Automation Conference. (2005)
25. Chu, M., Sulimma, K., Weaver, N., DeHon, A., Wawrzynek, J.: Object oriented circuit-generators in Java. In Pocek, K.L., Arnold, J., eds.: IEEE Symposium on FPGAs for Custom Computing Machines, Los Alamitos, CA, IEEE Computer Society Press (1998) 158–166