BCL: A Language for Hardware-Software Codesign

Authors' names removed for submission

Abstract

Special purpose hardware is vital to embedded systems as it can simultaneously improve performance while reducing power consumption. Due to time-to-market pressure, current design methodologies for embedded applications require an early determination of the design partitioning which allows hardware and software to be developed simultaneously, each adhering to a rigid interface contract. This approach is problematic because often a good hardware-software decomposition is not known until deep into the design process. Fixed interfaces and the burden of reimplementation prevent the migration of functionality motivated by repartitioning. We present the Bluespec Codesign Language (BCL), based on guarded atomic actions, to express the fine-grain parallelism necessary for both hardware and low-level software design. In this language, the programmer specifies the entire design, including the partitioning, allowing the compiler to synthesize efficient software and hardware, along with transactors for communication between the partitions. The benefit of using a single language to express the entire design is that a programmer can easily experiment with many different hardware/software decompositions. We present preliminary results for various hardware-software decompositions of an Ogg Vorbis audio decoder.

1. Introduction

Power and performance are increasing the need for hardware implementations of various components of embedded applications, such as video decoders and wireless transceivers. While a lot of effort is placed on the hardware components, equally important is the software driver which orchestrates the hardware blocks. Since the time-to-market is of paramount importance, the hardware and the associated software driver are almost always developed in parallel by two separate design teams. The two teams agree upon a hardware-software decomposition and the associated interface early on to make the final integration as seamless as possible. However, in practice the interface rarely matches the specification precisely. This happens because the early hardware specifications are often incomplete, leaving room for misinterpretation, or are simply unimplementable. This integration problem has a large negative impact on the time-to-market. Worse, by prematurely restricting the design, alternatives with lower costs (*i.e.,* area or power) or higher performance may be ignored.

We think that the fundamental difficulty of hardware-software codesign stems from the fact that the software and hardware continue to be developed in two separate languages, each with its own semantics and programming idioms. We believe that what is needed instead is a common language for hardware-software codesign with the following properties:

1. *Fine-grain parallelism:* Hardware is inherently parallel and any codesign language must be flexible enough to express meaningful hardware structures. Low-level software which drives the hardware does so via highly concurrent untimed transactions, which must also be expressible in the language.
2. *Easy specification of partitions:* In complex designs it is important for the designer to retain a measure of control in expressing his insights about the partitioning between hardware and software. Doing so within suitable algorithmic parameters should not require any major changes in code structure.
3. *Generation of high-quality hardware:* Digital hardware designs are usually expressed in RTL languages like Verilog from which low-level hardware implementations can be automatically generated using a number of widely available commercial tools. (Even for FPGAs it is practically impossible to completely avoid RTL). The codesign language must compile into efficient RTL code.
4. *Generation of efficient sequential code:* Since the source code is likely to contain fine-grain transactions, it is important to be able to interleave partially executed transactions without introducing deadlocks while waiting for external events.
5. *Shared communication channels:* Often the communication between a hardware device and a processor is accomplished via a shared bus. The high-level concurrency model of the codesign language should permit sharing of such channels without introducing deadlocks.

We are not aware of any language that has all the properties listed above. In spite of concerted efforts, hardware synthesis from sequential software languages like C, C++, Java has not been widely adopted for efficiency reasons. Hardware description languages (HDLs) like Verilog and VHDL provide extremely fine-grain parallelism but lack an understandable semantic model [1]. These HDLs are also impractical for writing software.

There are several parallel computation models whose semantics are agnostic to implementation in hardware or software. In principle, any of these can provide a basis for hardware-software codesign. *Threads and locks* are used extensively in parallel programming and also form the basis of SystemC [2] – a popular language for modeling embedded systems. However, good hardware synthesis from SystemC is unrealized for all but the most restrictive idioms. *Dataflow models*, both at macro-levels (Kahn [3]) and fine-grained levels (Dennis [4], Arvind [5]), provide many attractive properties but abstract away important resource-level issues that are important for expressing efficient hardware or software. Nevertheless dataflow models where the rates at which each node works are specified statically have been used successfully in signal processing applications [6, 7]. *Synchronous dataflow* is a clean model of concurrency based on synchronous clocks and forms the basis

of several programming languages (*e.g.,* Esterel [8], Rapide [9], Shim [10], Polysynchronoy [11]). We think that non-determinism is required to express hardware designs at a high-level and synchronous languages are not suitable for this.

We have chosen *guarded atomic actions* as the base for *BCL.* All legal behaviors in this model can be understood as a series of atomic actions on a state. This model was used by Chandy and Misra in Unity [12] to describe software, and then by Hoe and Arvind to generate hardware [13]. Dijsktra's Guarded Commands [14] and Lynch's IO Automata [15] are also closely related. BCL extends the idea of multiple clock domains [16] from Bluespec SystemVerilog (BSV) [17] to specify how a design should be split between hardware and software. Hardware compilation from BCL is a straightforward translation into BSV. Translation of BCL modules to efficient software is achieved using a new compiler which we have developed and will describe in this paper.

The primary contributions of this paper are the presentation of a single language which can be used to express fine-grain parallelism, a method of specifying hardware-software partitioning at the source level and a method of compilation of the software partitions into efficient software, including the transactors needed to communicate with the hardware partitions. We demonstrate the viability of our approach by applying it to a concrete example, an Ogg Vorbis audio decoder. We think this is the first demonstration of multiple versions of a hardware-software codesign from the same source code working on an FPGA .

Paper Organization: After a related work discussion, Section 3 presents the hardware-software codesign problem. In Section 4, we demonstrate some important features of BCL and its use in hardware design. Section 5 explains the details of the compilation strategy, and Section 6 demonstrates how partitions are specified in the language; these two sections can be read out of order. We conclude with a case-study, performance results, and a discussion.

2. Related Work

There is a substantial body of work, both academic and commercial, relevant to various aspects of hardware-software codesign.

1. Generation of software from hardware descriptions: Almost every hardware description language (HDL) can compile to a software simulator, which can be viewed as a software implementation. Popular commercial products like Verilator [18] and Carbon [19] show significant speedup in the performance of these simulators, though the requirement to maintain cycle-level accuracy (at a gate-level) is a fundamental barrier. The resulting performance is often several magnitudes slower than natural software implementations of the same algorithm.

Bluespec's Bluesim [17] can exploit the fact that the cycle-level computation can be represented as a sequence of atomic actions. This permits dramatic improvement in performance but the underlying cost of cycle-accuracy remains.

2. Generation of hardware from sequential software specifications: To ease the burden of hardware design, the idea of extracting a hardware design from a familiar software language, *e.g.,* C, Java, or various functional languages has great appeal. Many systems like CatapultC [20], Pico Platform [21], or AutoPilot [22] have been effective at generating some forms of hardware from C code. However, specifying efficient microarchitectures with dynamic control can be very hard, if not impossible, to construct using such tools [23].

A related effort is the Liquid Metal project [24] which compiles an extension of Java into hardware. It lets the programmer specify parts of the program in a manner which eases the analysis required for efficient hardware generation. In contrast to BCL which relies on explicit state and guarded atomic actions, Liquid Metal exploits particular extensions to the Java type system.

3. Frameworks for simulating heterogeneous systems: There are numerous systems that allow co-simulation of hardware and software modules. Such systems, which often suffer from both low simulation speeds and improperly specified semantics, are typically not used for direct hardware or software synthesis.

Ptolemy [25] is a prime example of a heterogeneous modeling framework, which concentrates more on providing an infrastructure for modeling and verification, and less on the generation of efficient software; it does not address the synthesis of hardware at all. Metropolis [26], while related, has a radically different computational model and has been used quite effectively for hardware/software codesign, though primarily for validation and verification rather than to synthesize efficient hardware.

SystemC [2], a C++ class library, is considered the most popular language to model heterogeneous systems. The libraries provide great flexibility in specifying modules, but SystemC lacks clear compositional semantics, producing unpredictable behaviors when connecting modules. Synthesis of high-quality hardware from SystemC remains a challenge.

Matlab and Simulink generate production code for embedded processors as well as VHDL from a single algorithmic description. Simulink employs a customizable set of block libraries which allow the user to describe an algorithm by specifying the component interactions. Simulink does allow the user to specify modules, though the nature of the Matlab language is such that efficient synthesis of hardware would be susceptible to the same pitfalls as C-based tools. A weakness of any library-based approach is the difficulty for users to specify new library modules.

In summary, while all these frameworks may be effective for modeling systems, we do not believe they solve the general problem of generating efficient implementations.

4. Algorithmic approaches to hardware/software partitioning: There is an extensive body of work which views hardware-software partitioning as an optimization problem, similar to the way one might look at a graph partitioning problem to minimize communication [27–29]. The success of such approaches depends upon the quality of estimates for various cost functions as well as the practical relevance of the optimization function. Since these approaches do not generate a working hardware/software design, they need to make high-level approximations, often making use of domain-specific knowledge to improve accuracy. Such analysis should be viewed as complementing real hardware-software codesign approaches.

3. Hardware-Software Codesign: an Example

As a running example in this paper, we employ Ogg Vorbis, an open-source psychoacoustically-aware audio CODEC aimed at simple low-complexity decoding. In this section, we give a brief description of the application, and introduce the codesign problem.

3.1 The Vorbis Pipeline

The input for Vorbis is a stream of frames of discrete compressed-audio samples and the output is a stream of PCM frames. An input frame typically contains a few thousand audio samples and some "configuration" information to specify sizes and encoding of the compressed data. All data in the Vorbis pipeline (see Figure 7) is passed through explicit channels at the frame granularity. We briefly describe the functionality of each block:

Stream Parser: This module is responsible for parsing configuration packets (Vector Quantization Tables and Huffman codebooks), and forwarding the appropriate configuration parameters to subsequent pipeline stages. It also directs the data to the correct decoder depending on the type of data.

Floor Decoder: This module reconstructs the spectral floor vector that has been encoded for each PCM channel. This vector is a

low-resolution representation of the audio spectrum for the current frame and is used as the baseline for spectral resolution.

Residue Decoder: The residue is the structure of the audio spectrum minus the floor curve. In this regard, it can be thought of as the error between the actual sound and the floor vector prediction. Residues are encoded using a lossy DCT compression.

IMDCT: This stage converts the full frame frequency spectrum, the result of summing the floor and residue vectors, to the time domain.

Windowing: After IMDCT we have a valid time-domain signal. However, our packetized encoding causes spectral leakage at frame boundaries. This is counteracted by overlapping consecutive frames via a sliding window function biased against the more error-prone parts of the reconstructed frame.

Conceptually we pass frames between stages of the Vorbis pipeline but in real designs (both hardware and software) frames are passed as a stream of smaller packets to facilitate pipelining.

3.2 The hardware-software codesign problem

The Stream Parser, and Floor and Residue decoders require minimal computation consisting mostly of data transfer and simple table lookups. Implementing these blocks in hardware is unlikely to produce much performance improvement or power savings. In contrast, IMDCT requires significant computation; internally, IMDCT uses an Inverse Fast Fourier Transform (IFFT) whose size varies dynamically from 128 to 8192 depending on the frame size. Windowing involves simple computation, but because it is directly connected to the PCM output it may make sense to implement it in hardware.

Given a frame of size N, IMDCT constructs a new frame of size $2N$ by applying two point-wise conversion functions and passes it into the IFFT.

The resulting frame is reordered (using bit-reversed indices), after which a point-wise function is applied to produce the final output. One could express IMDCT in pseudo C where `ifft` is the function to compute IFFT:

```
Array imdct(int N, Array vx){
 for(i = 0; i < N; i++){
   vin[i]   = convertLo(i,N,vx[i]);
   vin[i+N] = convertHi(i,N,vx[i]); }
 vifft = ifft(2*N, vin);
 for(i = 0; i < N; i++)
   vout[bitReverse(i)] = convertResult(i,N,vifft[i]);
 return vout; }
```

IFFT is usually computed using a mini IFFT of size K, called a *radix,* which is simply a K-input/K-output network of additions and multiplications. (The details of the radix are not important for our discourse). As an example, a 64-point IFFT using size 4 radices has 3 stages each containing 16 rows of radices. Each radix instance in a computation needs a set of rotation *omega* values which depend on the particular stage and row index of the radix in the larger computation. A radix is applied to K-elements of the input vector to produce K-elements of the output vector, however, the index of the output positions are not the same as the input indices. Hence the output indices have to be shuffled to be placed in the right place. The computation for an IFFT of size `sz` using K-size radices can be expressed as follows where `applyRadix` encapsulates the omega and index generation:

```
Array ifft(int sz, Array in){
 Array temp, sdata = in;
 int nStages = log(sz)/log(K), nRows = sz/K;
 for (int stage = 0; stage < nStages; stage++){
   for (int row = 0; row < nRows; row++)
     sdata = applyRadixK(sz, stage, row, sdata);
   for(int i = 0; i < sz; i++)
     temp[i] = sdata[permute[sz][i]];
   sdata = temp; }
 return sdata;
}
Array applyRadixK(int sz, int stage, int row, Array in){
 Array temp = in;
 temp[row:row+K-1]=radixK(in[row:row+K-1],
                          omega[sz][stage][row]);
 return temp;}
```

Now we illustrate some salient issues in hardware-software codesign using this example:

Hardware-software interface: For a compiler to create software which successfully calls the hardware IFFT it will have to give special treatment to instances of `applyRadixK` to correctly marshal information to and from the hardware-software communication channel. There are many choices to be made here. Do hardware and software work from a common memory? If so, all that must be done is place a pointer at the appropriate rendezvous. Or do we have to explicitly pass data through a memory mapped FIFO structure? These choices are going to vary from platform to platform and from one accelerator design to another.

Need for asynchronous request-response: There are a number of inefficiencies if we implement the hardware-software interface as described in the pseudo-code above. The software is sequential so the CPU will stall while an outstanding request is made, instead of constructing the next input to be fed to the hardware. Worse, this sequentiality prevents us from exploiting multiple accelerators or even just pipelining a single accelerator.

To have a truly high-performance design we have to expose more of the inherent parallelism of the hardware system. A simple way to do this would be to partition a call into an asynchronous request/response pair. The software can prime the pipeline with requests before reaching a steady state of alternating requests and responses. Unless specially designed to do so, this may still fail to handle any variance in latency in the underlying communication channel (*e.g.,* a shared bus) or in processing speed due to improvement/changes in the hardware accelerator. Proper exploitation requires a measure of time-dependent or non-deterministic choice.

Dynamic parameterization: IMDCT needs to compute IFFTs of sizes, *e.g.,* 128, 256, ... , 8192. Dynamic parameters have a very different effect on implementation in hardware and software. If the processor has only one adder and multiplier, the compiled code for all these IFFTs in software will be more or less the same. The compiled code may differ only because we may unroll some loops to take advantage of the ALU or memory pipelining. Therefore, the cost of a dynamic parameterization in software is minimal.

In hardware design, however, we can instantiate many adders, multipliers, registers etc., and are only constrained by the cost (area) and performance requirements. Typically one would implement even a fixed size IFFT quite differently depending upon the required performance. For example, a high performance 64-point IFFT may use 48 size-4 radices. A low cost implementation, on the other hand, might require us to trade time for space and use one size-4 radix repeatedly. This problem gets vastly aggravated if IFFTs of various sizes have to be computed. It is not likely to be practical to design many fixed size IFFTs and call the appropriate one as needed. Consequently, one would implement some fixed size IFFT in hardware and repeatedly call it from a software driver or a hardware FSM.

A panacea for hardware-software codesign would be a compiler which given a sequential program and some performance, cost, and power constraints, could automatically identify the blocks which need to be implemented in hardware, implement the selected blocks efficiently, and transform the code to make use of the new hardware blocks. BCL provides a part of the solution to this problem by providing a system where it is easy for the designer to construct many different hardware-software variants from one source description. It permits a natural representation of desirable hardware structures,

m ::=	`Module` mn $[t]$	// name, arg list
	$[n \leftarrow mn\ [v]]$	// instantiate state
	[`Rule` n a]	// Rules
	[`ActMeth` n $\lambda t.a$]	// Action methods
	[`ValMeth` n $\lambda t.e$]	// Value methods
v ::=	c	// Constant Value
‖	t	// Variable Reference
a ::=	r `:=` e	// Register update
‖	**`if`** e **`then`** a	// Conditional action
‖	a `|` a	// Parallel composition
‖	a `;` a	// Sequential composition
‖	a **`when`** e	// Guarded action
‖	$(t$ = e **`in`** $a)$	// Let action
‖	**`loop`** e a	// loop action
‖	**`loopGuard`** e a	// loopGuard action
‖	$m.g(e)$	// Action Methcall g of m
e ::=	r	// Register Read
‖	c	// Constant Value
‖	t	// Variable Reference
‖	e op e	// Primitive Operation
‖	e `?` e `:` e	// Conditional Expression
‖	e **`when`** e	// Guarded Expression
‖	$(t$ = e **`in`** $e)$	// Let Expression
‖	$m.f(e)$	// Value Methcall f of m
‖	**`loop`** $e1$ $e2$	// loop expression
op ::=	`&&` | `||` | ...	// Primitive operations
pr ::=	$[m]$ $(mn$,$[c])$	// BCL program

Figure 1. BCL Grammar without types

with software oriented methods of communication. Instead of providing an automatic way to partition the code, the BCL compiler makes it possible for the designer to evaluate many different partitioning in a real context.

4. Expressing Hardware in BCL

The syntax of BCL sans types is given in Figure 1; its semantics is borrowed from Bluespec and can be found in [30]. Here, we informally explain BCL's syntax and semantics via examples.

BCL is a language for hardware-software codesign, so we must have the flexibility to compile any BCL program into either hardware or software. Any language with a hardware target has to be restricted so that it is compilable into an efficient FSM. It is for this reason that BCL is best suited for writing software which requires no dynamic heap storage allocation and where the stack depth is known at compile-time. In spite of this restriction, BCL is a modern statically-typed language with higher-order functions and rich data structuring facilities to express fine-grain parallelism.

In this section we will show how IMDCT can be written in BCL to bring out various concurrency issues which are of interest to a hardware designer. Similar issues arise when software interfaces with hardware. In the next section we will show how a BCL design with such fine-grain parallelism can also be compiled into efficient software. We postpone the discussion of how the designer indicates which parts of his design should be implemented in hardware and which in software until Section 6.

Unlike sequential languages, a BCL program consists of explicitly declared state elements and a set of *guarded atomic actions* or *rules* on these state elements. The set of rules can be applied in any order on the state elements but only one at a time. Since the rule to be applied can be chosen arbitrarily, a BCL program can potentially generate non-deterministic behaviors. The reason we have chosen guarded atomic actions as the basis for our language is that this degree of nondeterminism is very useful in allowing a compiler to choose an particular implementation from a whole range of implementations. The compilation of such descriptions into efficient hardware is by now a well understood and a mature technology [13, 17]. A proper operational semantics for a language with guarded atomic actions (*i.e.,*the kernel of Bluespec) can be found in [30]; it provides a good foundation to build analysis and verification tools, as well the software compiler presented in this paper.

4.1 Preliminaries: Expressions, State Elements, and Rules

A combinational structure is a directed acyclic graph (DAG) of primitive functions like adders, multiplier, or simply Boolean operators at a lower level. In BCL, all such structures can be described as pure expressions. For example, for a given `K`, `applyRadixK` and `radixK` described in the previous section, are pure combinational functions. Their DAGs can be obtained by unrolling all loops and inlining all function calls. The arrays `sdata` and `temp` should be thought of as a bunch of wires, as opposed to storage elements. Since all array indices in this example are computable statically, each array operation simply becomes a wire connection. Syntactically `applyRadixK` and `radixK` would look the same in BCL!

Though `ifft` looks almost the same as these other functions, it will have be coded quite differently in BCL because we cannot unroll the loop with the dynamic parameter `sz`. Instead, we keep the dynamic data in registers – `sz`, `stage`, `row`, `sdata` and modify these using a rule. Each firing of the rule corresponds to one iteration of the inner loop in our C code. Each iteration reads and computes `K` elements; we use `sdata` to hold the newly computed `K` values in the places from where the old values are read. At the end of one stage of computation, `sdata` holds only newly created values which are then shuffled into the correct places.

```
rule doIFFT when(busy && (stage<nStages || row<nRows)){
  Vector#(max,t) temp = applyRadixK(sz,stage,row,sdata);
  if(row != nRows-1){row := row+1; sdata := temp;}
  else{stage := stage+1; row := 0;
       for(int i = 0; i < sz; i++)
          sdata[i] := temp[permute[sz][i]];}}
```

The rule has a predicate guard which must be true for the rule to execute. The busy flag, which we will describe later, signifies that there is an IFFT computation in flight. The `stage` and `row` registers play the role of loop indices and are set on each rule execution. The syntax (`r:=e`) indicates that register `r` is set to `e`. (A register assignment should not be confused by a variable binding which is written as (`r=e`)). Once enabled, this rule will execute repeatedly until the IFFT is completed at which point the guard will become false. As discussed before, the for-loop to shuffle data is unrolled statically in a hardware implementation and results in a wiring permutation.

We can easily translate this rule into hardware by constructing combinational circuits which evaluate the rules guard and body and add logic to update the relevant state when the guard is true. In fact C-to-hardware synthesis tools can also generate the same FSM automatically. However, in hardware this is not the only possible implementation; one might choose a different microarchitecture depending on the cost-performance tradeoff. A high-performance design may use multiple copies of `radixK` as well as introduce several stage buffers to operate in a fully pipelined manner. Interested readers can find a detailed discussion of many variants in the context of a 802.11a CODEC implementation in [31]. These variants are too different from each other to be generated in an automatic manner from the same C code. Some of these variants are also practically impossible to express in a sequential language [23].

In general multiple rules on the same state elements are used to describe the desired behavior. The BCL semantics only requires that one rule be executed at a time and the rule to be executed can be chosen arbitrarily. This choice introduces the nondeterminism often needed to express hardware at a more abstract level. For performance reasons, the Bluespec compiler does static analysis of the rules and schedules as many rules to execute concurrently as possible without violating the one-rule-at-a-time semantics [13]. The compiler essentially generates a circuit for each rule and synthesizes a scheduler for the correct execution of multiple rules. We omit a more complete description of hardware compilation because

BCL compiler simply calls the Bluespec SystemVerilog [17] compiler once the hardware partition has been isolated.

4.2 Packaging IFFT into a Module

In BCL, a module is the basic linguistic abstraction to encapsulate computation and state. Like an object-oriented language, one thinks of a module as an object with method interfaces; the object's state can be manipulated only via the object's methods. However, the methods in BCL are different in one important aspect: each method has an implicit *guard* and a method can be called only when its guard is true. Consequently a rule is ready to execute only when the rule's guard and all the methods called by the rule are ready. Such guarded methods provide a convenient way to build large atomic actions in a modular way and gives us flexibility to refine the timing of internal modules.

To package the IFFT from the previous section into a module we need to consider the interface through which the external world will interact with the IFFT. Logically, we need to receive an input frame and send out the resulting output frame. However, we do not want to make any assumptions about how long the computation takes and therefore split the IFFT call into request and response methods. Since we want the design to be implementable in hardware we cannot pass varying sized frames into our IFFT. Instead we will fix the size to the maximum supported frame size, (pad the input and output) and add an additional integer parameter to represent the size of the current frame. This is captured in the following interface definition parameterized by the data type `t`:

```
interface IFFT#(type t){
 method Action input(int sz, Vector#(max,Complex#(t)) x);
 method ActionValue#(Vector#(max,Complex#(t))) output();}
```

The `input` method is an `Action`, *i.e.,* it causes a state change when it executes but it returns nothing. Specifically, it sets the loop counters, data and flag registers to initialize the computation. Methods which only read state and return a result based on it are pure functions and called *value methods*.The `output` method is an `ActionValue` method which performs both an action on the state and returns a value. In this case `output` returns the completed result from the IFFT and marks it as having been removed. The designation of methods as Action or ActionValue is important since the BCL type system employs monadic isolation all mutations, easing the burden of analysis and guaranteeing the safety of pure functions. We wrap our rule in a module as follows:

```
module mkIFFT(IFFT){
 Reg#(int)             sz <- mkRegU();
 Reg#(bool)          busy <- mkReg(False);
 Reg#(int)          stage <- mkReg(0);
 Reg#(int)            row <- mkReg(0);
 VectorReg#(max,t) sdata <- mkVectorReg();
 let nStages = tabulate(\s -> log(s)/log(k))[sz]);
 let nRows   = sz/K;
 rule doIFFT ...
 method Action input(int s, Vector#(max,t) x)when(!busy){
     stage:=0; row:=0; busy:=True; sdata:=x; sz:=s;}
 method ActionValue#(Vector#(max,t)) output()
       when (busy && stage == nStages && row == nRows){
     busy := False; return sdata;}}
```

Notice that our method instances have `when` clauses like our rule. Here it serves the same purpose, to signify when those methods are valid to be called. This condition will be folded into the guard of any rule calling this method.

4.3 IMDCT: Using the IFFT Module

The polymorphic code for IMDCT is given in Figure 2. Its `input` method setups the input for the IFFT and calls the `input` method of the IFFT, while the `output` method extracts the output from IFFT. The syntax of the first line in Figure 2 can be read as follows: "a module named `mkIMDCT` being defined. It takes the module `mkIFFT` (of type `IFFT#(t)`) as a constructor argument, and implements the interface `IMDCT#(t)`".

```
module mkIMDCT#(Module#(IFFT#(t))) mkIFFT)(IMDCT#(t)){
 IFFT#(Complex#(t))  ifft <- mkIFFT;
 FIFO#(Vector#(2*MAX,Complex#(t)) queue <- mkFIFO;
 method Action input(vx) {
   Vector#(2*MAX,Complex#(t)) v;
   for(int i = 0; i < K; i++){
     v[i]   = preTable1[i]*vx[i];
     v[K+i] = preTable2[i]*vx[i];}
   ifft.in(2K,v); }
 method ActionValue#(Vector#(2*MAX,Complex#(t)))
  output(){
   let x <- ifft.output();
   Vector#(2K,Complex#(t)) v;
   for(int i = 0; i < 2*K; i++)
     v[i] = x[bitReverse(i)];
   return v;}
```

Figure 2. mkIMDCT module definition

4.4 Marshaling/Demarshaling

All the interfaces we have seen thus far receive input and return output as a single vector. Implementing these blocks in hardware requires prohibitive routing resources, and this cost only becomes more obvious when one considers the additional waste incurred for dynamic frame sizes, since we must always pay for the maximum size. Additionally, this style doesn't match physical channels between hardware and software, which are generally data width limited. With these considerations, a more efficient streaming interface for the IFFT might be written as follows:

```
interface IFFT#(type t){
 method Action setSize(int sz);
 method Action input(t x);
 method ActionValue#(t) output();}
```

Using the new interface, the dynamic size is set using the `setSize` Action method, after which the invoking rules can stream data in and out of the module using the `input` and `output` methods. Once a full frames worth of elements have been streamed in, the input methods guard protects the buffered data from being overwritten while `doIFFT` performs the computation "in place". Once the user has read the transformed frame, the input method can again begin to receive data.

We can convert the IFFTs presented in this section to use this interface by inserting marshaling and demarshaling code in the input/output methods, as shown below for the input case:

```
module mkIFFT(IFFT){
 ...
 Reg#(int) in_cnt <- mkReg(0); //added state
 VectorReg#(max,t) in_buff <- mkVectorReg();
 method Action input(t x) when (!busy || in_cnt+1 < sz){
   in_buff[in_cnt] := x;
   if(in_cnt+1 < sz) {in_cnt := in_cnt+1;}
   else{sdata := in_buff; in_cnt := 0; busy := True}}
 ... }
```

This new implementation not only synthesizes to much more efficient hardware, but lets us hide some of the latency of the `doIFFT` rule by buffering up the next frame while the current one is in flight. Since BCL is a hardware-software codesign language, it must also be possible to generate efficient software from BCL; we will describe this in the next section. A reader more interested in understanding partitioning can read Section 6 before reading Section 5 without loss of continuity.

5. Compiling BCL into software

Each BCL module is compiled into a C++ class and each of the module's rules and methods is compiled into a separate class method. During the course of translation we need to create shadow

copies of the state of an object. This is accomplished by constructing a new object of the same class using the copy constructor. The translation procedure maintains an environment ρ which maps BCL module instance names to the active class instance corresponding to that name. A new ρ containing all immediate submodules is constructed for the compilation of each BCL module, and serves as the *initial* environment for the translation of each internal rule and method. We now present a detailed syntax directed compilation of BCL. The compiling scheme is presented bottom up, starting with the compilation of expressions then actions and rules, followed by modules. We finally discuss some optimizations to improve the code efficiency.

For the sake of brevity we take a few notational liberties in describing translation rules. Generated C++ code is represented by the conjunction of three different idioms: literal C++ code (given in the `true text font`), syntactic objects which evaluate to yield C++ code (given in the document font), and environment variables used by the compiler procedures (represented as symbols). The names of compiler procedures that generate code fragments are given in **boldface**.

5.1 Compiling Expressions

The translation of a BCL expression produces a C++ expression and one or more statements that must be executed before the expression. The procedure to translate expressions (**TE**) is shown in Figure 3; it is straightforward because expressions are completely functional. The only clause needing further explanation is the guarded expression (e **when** ew). Upon evaluation, the failure of a sub-expression's guard should cause the entire expression to evaluate to $\perp$. If the value of an expression evaluates to $\perp$, then its use in an action causes that action to have *no effect*. The "throw" in case of guard failure is "caught" in the lexically outermost action enclosing the expression.

TE :: **Env** × ⟦ e ⟧ → (CStmt, CExpr)

TE ρ ⟦ r ⟧ = (;, ρ[r].read())

TE ρ ⟦ c ⟧ = (;, c)

TE ρ ⟦ t ⟧ = (;, t)

TE ρ ⟦ e1 **op** e2 ⟧ = (s1;s2, ce1 **op** ce2)
where (s1, ce1) = **TE** ρ ⟦ e1 ⟧
(s2, ce2) = **TE** ρ ⟦ e2 ⟧

TE ρ ⟦ ep **?** et **:** ef ⟧ = (sp; st; sf, cep `?` cet `:` cef)
where (sp, cep) = **TE** ρ ⟦ ep ⟧
(st, cet) = **TE** ρ ⟦ et ⟧
(sf, cef) = **TE** ρ ⟦ ef ⟧

TE ρ ⟦ e **when** ew ⟧ = (se; sw; `if (!cw) { throw GuardFail; }`,ce)
where (se, ce) = **TE** ρ ⟦ e ⟧
(sw, cw) = **TE** ρ ⟦ ew ⟧

TE ρ ⟦ t = et **in** eb ⟧ = (st; t = ct; sb, cb)
where (st, ct) = **TE** ρ ⟦ et ⟧
(sb, cb) = **TE** ρ ⟦ e ⟧

TE ρ ⟦ m.f(e) ⟧ = (se, ρ[m]`.f(ce)`)
where (se, ce) = **TE** ρ ⟦ e ⟧

Figure 3. Translation of Expressions

5.2 Compiling Actions

A rule is composed of actions and any of these actions can have guards. Earlier we explained the meaning of a guarded action by saying that a rule is not eligible to fire (execute) unless its guard evaluates to true. However, due to conditional and sequential composition of actions, in general it is impossible to know if the guards of all the constituent actions of a rule are true before we execute the rule. To circumvent this limitation, we execute a rule in three phases: In the first phase we create a shadow of all the state elements using the `copy` constructor. We then execute all constituent actions, updating the shadow state. Sometimes more shadows are needed to support the parallel semantics within an action. (In our translation rules, we only copy and create new environments and let the generated C++ code mutate the objects referenced by the environment). Finally, if no guard failures are encountered we commit the shadows, that is, atomically update the real state variables with values of the shadowed state variables using `parMerge` or `seqMerge` depending upon the action composition semantics. On the other hand if the evaluation encounters a failed guard, it aborts the computation and the state variables are not updated.

For perspicuity, the rules present a slightly inefficient translation where shadows of the entire environment are created whenever a shadow may be needed. Figure 4 gives the procedure for translating BCL actions (**TA**).

TA :: **Env** × ⟦ a ⟧ → CStmt

TA ρ ⟦ r **:=** e ⟧ = se; ρ[r]`.write(ce);`
where (se, ce) = **TE** ρ ⟦ e ⟧

TA ρ ⟦ **if** e **then** a ⟧ = se; `if (ce) {`**TA** ρ ⟦ a ⟧`}`
where (se, ce) = **TE** ρ ⟦ e ⟧

TA ρ ⟦ a1 | a2 ⟧ = cs1; cs2; (**TA** ρ1 ⟦a1⟧); (**TA** ρ2 ⟦a2⟧); pm; ms;
where (cs1, ρ1) = **makeShadow** ρ
(cs2, ρ2) = **makeShadow** ρ
(pm, ρ3) = **unifyParShadows** ρ1 ρ2
ms = **commitShadow** ρ ρ3

TA ρ ⟦ a1**;**a2 ⟧ = cs; (**TA** ρ1 ⟦a1⟧); (**TA** ρ1 ⟦a2⟧); ms;
where (cs, ρ1) = **makeShadow** ρ
ms = **commitShadow** $\rho\rho$1

TA ρ ⟦ a **when** e ⟧ = se;`if(!ce){throw GuardFail;};`ca
where (se, ce) = **TE** ρ ⟦ e ⟧
ca = **TA** ρ ⟦ a ⟧

TA ρ ⟦ t = e **in** a ⟧ = se; `t = ce;` (**TA** ρ ⟦ a ⟧)
where (se, ce) = **TE** ρ ⟦ e ⟧

TA ρ ⟦ m.g(e) ⟧ = se; (ρ[m]`.g(ce);`
where (se, ce) = **TE** ρ ⟦ e ⟧

TA ρ ⟦ **loop** e a ⟧ = cs; `while(true){se; if(!ce) break; ca;}` ms;
where (cs,ρ1) = **makeShadow** ρ
(se, ce) = **TE** ρ ⟦ e ⟧
ms = **commitShadow** ρ ρ1
ca = **TA** ρ1 ⟦a⟧

TA ρ ⟦ **loopGuard** e a ⟧ = `try{while(true){ cs; se; if(!ce) break; ca; ms;}}catch {}`
where (cs,ρ1) = **makeShadow** ρ
(se, ce) = **TE** ρ ⟦ e ⟧
ms = **commitShadow** ρ ρ1
ca = **TA** ρ1 ⟦a⟧;

Figure 4. Translation of Actions

State Assignment ($r := e$): This causes a side-effect in the relevant part of the state of the object which can be extracted from ρ. If e evaluates to bottom, the control would have already been transferred automatically up the call stack via the `throw` in e.

Parallel Composition ($a1 \mid a2$): Both $a1$ and $a2$ observe the same initial state, though they update the state separately. Consider the parallel actions $r_1 := r_2 | r_2 := r_1$ which swap the values of r_1 and r_2. Such semantics are naturally implemented in hardware as swaps can be done with no intermediate state (the values are read in the beginning of a clock cycle and updated at the end of it). However, in software if we update r_1 before executing the second action, then the second action will read the new value for r_1 instead of the old one. To avoid this problem, the compiler creates shadow states for each parallel action, which are subsequently merged after both actions have executed without guard failures. In a legal program, the updates of parallel actions must be to disjoint state elements. Violation of this condition can only be detected dynamically, in which case an error is thrown.

The compiler uses several procedures to generate code to be used in implementing parallel composition. The **makeShadow** pro-

cedure takes as its argument an environment (ρ) and returns a tuple consisting of a new environment (say $\rho 1$), and C++ statements (say cs1). cs1 is executed to declare and initialize the state elements referenced to in $\rho 1$. The new environments are then used in the translation of each of the actions. The procedure **unifyParShadows** is used to unify $\rho 1$ and $\rho 2$, implicitly checking for consistency. Along with $\rho 3$, which contains the names of the unified state elements, it returns a C++ statement (pm) which actually implements the unification. Lastly, the **commitShadow** procedure generates code (ms) to commit the speculative state held in $\rho 3$ back into the original state ρ.

In order to understand **unifyParShdows**, consider the parallel merge of two primitive Registers, each of which track their own modification with a dirty bit:

```
void parMerge(Reg<T>& r1, Reg<T>& r2){
  if(r1.dirty && r2.dirty) {throw ParMergeError;}
  if (r2.dirty) {r1.val = r2.read(); r1.dirty = true};
```

From here, the reader can extrapolate the implementation to other primitive modules such as the `VectorReg` used in the IFFT implementation. The `parMerge` for this module would most likely allow updates to disjoint locations in parallel branches of execution, but throw an error if the same location were written. **unifyParShadows** generates code which recursively invokes `parMerge` pointwise on the two environments, whereas **commitShadow** ρ $\rho 1$ simply performs pointwise updates of objects in ρ from dirty objects in $\rho 1$ by invoking `seqMerge`.

Sequential composition ($a1; a2$): The action $a1; a2$ represents the execution of $a1$ followed by $a2$. $a2$ observes the full effect of $a1$, but due to the atomic nature of action composition, no one can observe $a1$'s updates without also observing $a2$'s. The subtlety in sequential composition is that if $a1$ succeeds but $a2$ fails, we need some way of undoing $a1$s updates. Its because of this that we need to create a shadow state before executing either action. As the sequential rule in Figure 4 shows, after creating the shadow $\rho 1$, we pass it to the translation of both $a1$ and $a2$. The code block ms commits the resulting state. However $a1; a2; a3$ can all be executed using only one shadow.

Guarded Action (a when e): The **when** keyword allows the programmer to guard any action. The C++ translation of guarded actions executes the guard, se, followed by the expression ce, and throws a guardFail exception if ce evaluates to false. After this, the code implementing a is executed.

Atomic Loop (loop e a): The semantics of this loop can be understood as the reduction: **loop** e a $\Rightarrow$ **if** e **then** $(a;$ **loop** e $a)$ **else** $noAction$. In other words all iterations are sequentially composed to form one atomic action. Consequently, if any iteration fails, the whole loop action fails. This can be implemented using one shadow which is created before entering the loop.

Protected Loop (loopGuard e a): The difference between the protected loop and the atomic loop are in the termination semantics. In protected loop, if a guard in a loop iteration fails, that *iteration* is treated as `noAction`. In the atomic loop, the entire loop action becomes `noAction`.

The reader might notice the absence of a parallel loop. This is not due to any fundamental barrier in the semantics, but rather the authors' prejudice in how this affects design patterns in BCL.

5.3 Compiling rules and methods

Figure 5 gives the translation of rules and methods, differentiating between action and value methods. The important thing to note is that rules catch guard failures, whereas methods do not. This is consistent with our implementation of the BCL semantics that specify rules as *top level* objects which cannot be invoked within the language. Methods, on the other hand, must be called from inside a rule or another method.

genRule ρ ⟦ (Rule nm a) ⟧ =
  `void nm(){try{`**TA** ρ ⟦ a ⟧`}catch{guardFail};}`
genAMeth ρ ⟦ (AMeth nm v a) ⟧ = `void nm(t v){`**TA** ρ ⟦ a ⟧`}`
genVMeth ρ ⟦ (VMeth nm v e) ⟧ = let (se,ce) = **TE** ρ ⟦ e ⟧
  in `void nm(t v){se;return ce;}`

Figure 5. Translation of Rules and Methods

5.4 Compiling Modules and generating `main()`

The C++ class corresponding to a BCL module has five methods in addition to a method for each BCL modules rules and methods. These five methods are a default constructor (used to instantiate its submodules recursively), a copy constructor (used to generate shadows, also recursive), `parMerge`, `seqMerge`, and a method to execute internal rules.

Each BCL module instantiates all of its submodules and specifies their initial state. The corresponding C++ default constructor implements this behavior exactly. Thus, by instantiating the top module, the entire program state is recursively instantiated. For the translation of the methods and rules of each module definition, an environment ρ is constructed to refer to these newly instantiated objects. This environment is used to compile all rules and methods of this module as shown in Figure 5.

The execution semantics of a BCL program is defined by the firing of its constituent rules. While each rule modifies the state deterministically, nondeterminism is introduced via the choice of which ready rule to execute. The range of behaviors that a collection of rules and state can produce is described below:

Repeatedly:
- Choose a rule to execute
- Compute the set of state updates in a shadow state U, by evaluating the rule's action according to the rules given in [30].
- Commit the shadow U to the original state.

This nondeterminism is resolved in each module by its method, `execSchedule`. There are many important scheduling choices, which greatly impact execution performance. A trivial schedule simply iterates through each rule in a module, before recursively invoking `execSchedule` on all submodules one by one. Once the `execSchedule` of each module has been invoked recursively, we re-invoke `execSchedule` on the top module. We will discuss improvements to this approach to scheduling in Section 5.5.

Since BCL is not sufficiently dynamic to describe software at higher levels in the software stack, facilities have to be provided to interface BCL with other languages. We present an interface of guarded methods to software, and an application seeking to use BCL as a slave can directly call these methods after first verifying that a method's guard is "ready". If a peer-oriented view is desired, the BCL design may be required to call directly into the external application. This requires designers to present an appropriate guarded interface to the BCL application.

5.5 Optimizations

The most effective means of increasing software execution speed is the minimization of shadows. There several ways to accomplish this goal:

Sequencing Parallel Actions: The conversion of a parallel composition to a sequential one results in the saving of one shadow copy. For example, by converting $r1 := f(x) \mid r2 := g(x)$ to $r1 := f(x)$; $r2 := g(x)$, we need not shadow states $r1$ and $r2$ twice before executing the parallel branches. In addition, action methods invoked on the same module in parallel which internally modify disjoint states may also be sequentialized. Even sim-

ple data-flow analysis can reveal abundant opportunities for this optimization.

Lifting Guards: When executing a rule whose guard fails, we would prefer it fail as early in the execution as possible. Early failure avoids all sorts of extra work, most importantly shadowing. Consider the following transformation in BCL:

$(a1$ **when** $a1_g \mid a2$ **when** $a2_g) \Rightarrow (a1 \mid a2)$ **when** $(a1_g \wedge a2_g)$

A C++ implementation of these semantics requires guards be evaluated before either a1 or a2 is executed. By the rules given in [30], it is possible to lift most guards.

Scheduling: The first order concern in scheduling is choosing a rule which will not fail, since partial execution of any rule which eventually fails is wasted work. Scheduling annotations in the form of priorities among rules can be given by the programmer to direct the compiler'scheduling decisions. Even without such annotations, the compiler can exploit data-flow analysis which may reveal that the execution of one rule may enable another. Portions of BCL programs are sometimes identifiable as synchronous regions in which case a well-known static scheduling techniques are applied.

Partial Shadowing: The reference implementation constructed a complete shadow for every possible action composition. If compiler analysis reveals that only a subset of the state can be modified by an action, only that subset needs to be shadowed.

The software implementation presented in this paper is single-threaded, but it can be extended to multi-threaded execution environment to exploit parallelism. Work on this front ongoing, but beyond the scope of this paper.

6. Specifying Partitions

We return to the original IMDCT example to demonstrate one of the most important features of BCL: *specifying partitions*. As the first example of partitioning, we will implement the IFFT module in hardware and the remaining functionality in software. Later we will discuss various partitioning of the complete Vorbis decoder.

6.1 Computational Domains

Partitions are specified using computational domains, and a type system is used to enforce that partitions are disjoint and no inadvertent inter domain references are made. Imagine annotating every method with a domain name and following the simple discipline that each rule (and method) can refer to methods of only one domain. Consequently, if a rule (or method) refers to methods from domain D only then we can say that the rule belongs to domain D. A simple type checker can determine if the domain annotations are consistent. A primitive module, *i.e.,* one that does not refer to the methods of any external module, is said to be a *synchronizer* if it has methods in more than one domain. Interdomain communication is possible only through synchronizers. In fact, as we show below inserting synchronizers in a program is the same as specifying a partitioning of that program.

Consider the IMDCT code shown in Figure 2. Suppose we want to implement the IFFT in hardware and the rest of IMDCT in software. The input and output methods of IFFT must be in the hardware domain (HW) but the IMDCT methods that invoke these methods must be in the software domain (SW). This seems impossible unless we violate our type discipline which says the domain of a method must be the same as the domain of the methods it calls. The way we resolve this problem is by introducing two synchronizing FIFOs with the following interfaces:

```
interface SyncFIFOHtoS#(type t){
 (HW) Action enq(t val);
 (SW) ActionValue#(t) deq(); }
```

```
interface SyncFIFOStoH#(type t){
 (SW) Action enq(t val);
 (HW) ActionValue#(t) deq();}
```

These synchronizing FIFOs can be inserted in the original IMDCT code of Figure 2 by introducing two new rules in the HW domain to talk to the IFFT and modifying each method of IMDCT to call the appropriate synchronizing FIFO rather than the IFFT directly. This is given in Figure 6.

```
module mkIMDCT#(Module#(IFFT#(Complex#(t))) mkIFFT)
                                          (IMDCT#(t)){
 IFFT#(Complex#(t)) ifft <- mkIFFT;
 FIFO#(Vector#(2*MAX,Complex#(t)) queue <- mkFIFO;
 SyncFIFOStoH#(int)                        sizeSync <-
  mkS2H;
 SyncFIFOStoH#(Vector#(2*MAX, Complex#(T)))  inSync <-
  mkS2H;
 SyncFIFOHtoS#(Vector#(2*MAX, Complex#(T))) outSync <-
  mkH2S;
 rule feedIFFT{ // HW
   let v <- inSync.deq();
   let s <- sizeSync.deq(); ifft.input(s,v);}
 rule drainIFFT{ // HW
   let x <- ifft.output(); outSync.enq(x);}
 method Action input(vx) { // SW
   Vector#(2*MAX,Complex#(t)) v;
   for(int i = 0; i < K; i++){
     v[i]   = preTable1[i]*vx[i];
     v[K+i] = preTable2[i]*vx[i];}
   inSync.enq(2K,v); }
 method ActionValue#(Vector#(2*MAX,Complex#(t)))
  output(){ // SW
   let x <- outSync.deq();
   Vector#(2K,Complex#(t)) v;
   for(int i = 0; i < 2*K; i++)
     v[i] = x[bitReverse(i)];
   return v;}
```

Figure 6. partitioned mkIMDCT module definition

Though moving the design from a single domain to multiple domains required modifying only a few lines to code, the effect is to introduce buffering along the hardware/software cut of the data-flow graph. In general such buffering can affect the functionality of the design. However if we restrict such changes to interfaces that are *latency-insensitive*, then such changes are correct by construction. It is beyond the scope of this paper to define this property precisely, but it suffices to say that latency sensitive hardware-software interfaces are usually quite brittle and hence of little use. When using BCL for hardware software codesign, an effective design methodology requires all hardware/software interfaces to be latency insensitive. We note in passing that it is this property of interfaces which enables modular refinement of a design: in fact, moving a module from hardware to software *is* modular refinement [32].

6.2 The Complete Vorbis Pipeline

The Vorbis pipeline can be partitioned in many different ways; six are shown in Figure 7. We present the code corresponding to the partition with the IFFT and windowing function placed in hardware. The PCM output is naturally in hardware (speaker), so we don't require an additional rule to stream the output of the windowing function back to software.

```
module mkVorbisPipeline(VorbisPipeline){
 SOURCE source <- mkVorbisStreamGen();
 Parser parser <- mkStreamParser();
 ResDec resdec <- mkResidueDecoder();
 FloDec flodec <- mkFloorDecoder();
 IMDCT   imdct <- mkIMDCT;
 SyncFIFOStoH#(Complex#(t)) toWindow <- mkStoH;
 WINDOW window <- mkWindowingFunction();
 rule sourceToParser  ... // SW
 rule parserToFloor   ... // SW
 rule parserToResidue ... // SW
```

```
rule reconstSignal{      //SW
  let x <- resdec.output();
  let y <- flodec.output();
  imdct.input(x+y);}
rule imdctToSync    ... // SW
rule syncToWindow   ... // HW
```

It is possible to write very general partitioned code where domains are represented as type parameters. We provide the following primitive to enable this level of flexibility:

```
interface Sync#(type t, domain a, domain b){
 (a) Action enq(t val);
 (b) ActionValue#(t) deq();}
```

Suppose we declare the IMDCTs internal synchronizers in the following manner, where **a** is a free type variable:

```
Sync#(int,         a, HW) sizeSync <- mkSync;
Sync#(Complex#(t), a, HW) inSync   <- mkSync;
Sync#(Complex#(t), HW, a) outSync  <- mkSync;
```

The resulting IMDCT is fully polymorphic in its domain type. If the parameter **a** is instantiated to HW, the compiler will recognize that the synchronizers are hardware to hardware and will replace them with lightweight FIFOs. If on the other hand **a** is instantiated to SW then the synchronizers would become mkSyncHtoS (or StoH). In other words, a very general partitioned code may insert more synchronizers than necessary for a specific partitioning, but these can be optimized by the compiler in a straightforward manner.

6.3 Separating Hardware from Software: Generating Partitions

We can extract the code for a particular domain D by removing all the rules not annotated with D from the partitioned code. We do not have to remove methods since our type system guarantees that any method not in D cannot be invoked by the rules of D; such dead methods will be removed by the compiler in the target domain. Once separated, *each partition can now be treated as an distinct BCL program and compiled to its designated target domain in isolation*. Shown below is the BCL code corresponding to the `SW` partition of the Vorbis pipeline presented above. This code can now be compiled into software according to the rules in Section 5:

```
module mkVorbisPipelineSW(VorbisPipeline){
 SOURCE source <- mkVorbisStreamGen();
 Parser parser <- mkStreamParser();
 ResDec resdec <- mkResidueDecoder();
 FloDec flodec <- mkFloorDecoder();
 IMDCT   imdct <- mkIMDCT;
 Sync#(Complex#(t),SW,HW)   toWindow <- mkSync;
 WINDOW window <- mkWindowingFunction();
 rule sourceToParser  ... // SW
 rule parserToFloor   ... // SW
 rule parserToResidue ... // SW
 rule reconstSignal   ... //SW
 rule imdctToSync     ... // SW
```

6.4 Sharing the communication channel

A partitioned code often involves many synchronizers. For example, in this partitioning, we use three synchronizers from software to hardware and one in the other direction. Often systems will provide fewer physical channels (typically one in the form of a bus) which must be multiplexed to accommodate multiple virtual channels. It is possible to express this multiplexing in BCL, but it is easier for the designer to use the highly parameterized library modules provided for this purpose.

In the next section we evaluate a range of partitionings where in each case, we run the software partition either on a processor connected the FPGA with a PCI-E bus, or on a processor embedded in the FPGA fabric. In the embedded case, the communication channels between hardware and software are point-to-point and no multiplexing is required.

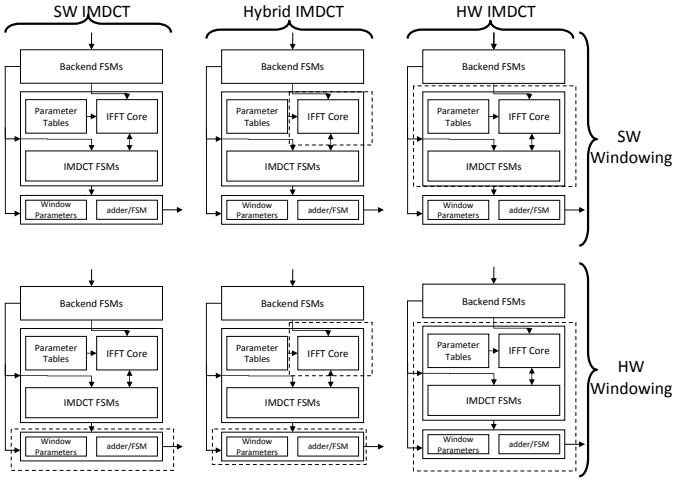

Figure 7. Examined partitions of Ogg Vorbis. The dotted section denotes the hardware partition.

7. The Experiment

In order to decice a hoog HW/SW partition. many compoennts must be very strong. Hardware must be good. THe software must be good, the communication must be good. The measurement methodology must be good. Besides the hardware quality, which we significant experience with

To show the effectiveness of BCL in codesign we explored six different hardware/software partitionings of the Vorbis decoder. These partitionings were derived by placing the IMDCT and windowing stages in either hardware or software. Three different designs for IMDCT were considered: a pure software IMDCT, a hardware IMDCT, and a hybrid IMDCT with a hardware IFFT orchestrated by a software FSM. (see Figure 7). As claimed by our methodology, each of these partitions was specified and compiled in minutes. We used Xilinx XUPv5-LX110T FPGA platform as the target for hardware compilation.

To evaluate a

We evaluated the software by running it on two different platforms. The first platform is a standard hardware acceleration platform, where XUPv5 is connected to a 2-core 2.8GHz Nehalem Westmere CPU with 3GB of RAM. Communication is done through PCI-Express on top of the Standard Co-Emulation and Modeling infrastructure (SCE-MI), a standard protocol geared towards testing and timing of hardware systems. The second platform represents a classic embedded system and uses a MicroBlaze softcore inside the XUPv5 FPGA. This embedded processor runs only at 100MHz but communicates with other FPGA objects using the Fast Simplex Link (FSL) protocol on uni-directional point-to-point communication channel. Running our software on these two platforms only required us to use a different communication module.

To evaluate performance, we constructed a testbench consisting of 10000 Vorbis audio frames. All computation was done using 32-bit fixed point values with 24-bit fractional precision. The performance results and FPGA hardware partitionings for both platforms are shown in Figure 8.

As expected, in the accelerator platform performance tends to improve when we place more computation on FPGA. We can also see that the cost of communication on the PCI-express is significant. So much so that placing partitions which increase communication (only hardware windowing versus pure software) causes a notable slow down despite the fact that the hardware windowing implementation is faster.

The embedded system has a much lower latency through its communication channel. However, the performance of software is

FrontEnd:	SW	SW	SW	SW	SW	SW
IMDCT:	Full HW		HW IFFT		Full SW	
Windowing:	HW	SW	HW	SW	HW	SW
	FPGA Accelerator Platform					
Speed(secs)	8.9	28.1	84.9	102	316	38.9
FPGA (Regs)	36%	32%	40%	36%	34%	0%
FPGA (Slices)	22%	22%	20%	21%	23%	0%
	Embedded FPGA Platform					
Speed(secs)	114	231	414	421	876	896
FPGA (Regs)	36%	35%	38%	37%	35%	33%
FPGA (Slices)	36%	35%	33%	34%	37%	39%

Figure 8. Execution and Simulation Results.

also reduced relative to the FPGA. As a result in the embedded system it's much more advantageous to consider systems with more hardware than in the case of the accelerator and we end up with the entire backend in hardware.

While this experiment handily shows how we can effectively migrate designs across platforms and do major architectural changes on a single platform. What remains to be seen is how does this implementation fare against a hand-coded design. To evaluate this we hand-coded the software partition for the best design partitioning for the embedded platform. As hardware quality of this methodology has been proven multiple times, we reuse the hardware for testing. The resulting code is 11% faster than our built design. Analysis shows that much of the efficiency loss is due to inefficient scheduling of the software rules. With more intelligent software rule scheduling we can get comparable efficiency. For designs where pointers/specialized implementations cause significant improvement.

8. Conclusion

We have presented a novel approach for exploring hardware-software codesign tradeoffs based on expressing the entire design using a language of guarded atomic actions. Using the example of a Vorbis decoder we've shown that partitioning of designs in BCL is natural and easy and can hide the messiness of the underlying communication substrate. We doubt if so many different hardware-software partitioning of Vorbis can be specified and evaluated in one day using any two language framework. We have shown that our compiling technique is platform independent by running it on two very different software platforms. We used high quality commercial tools for compiling hardware and showed that our software compilation generated code whose efficiency was comparable to hand-coded C. Our approach generalizes to multiple software threads and multiple hardware substrates naturally, though we did not explore that aspect in this paper. While it's unlikely that the first partitioning a designer attempts is optimal, BCL enables rapid refinements, and allows the designer to tune the design

References

[1] P. O. Meredith, M. Katelman, J. Meseguer, and G. Roşu, "A formal executable semantics of Verilog," in *Eighth ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE'10)*. IEEE, 2010, pp. 179–188.

[2] S. Y. Liao, "Towards a new standard for system level design," in *Proceedings of the Eighth International Workshop on Hardware/Software Codesign*, San Diego, CA, May 2000, pp. 2–7.

[3] G. Kahn, "The semantics of simple language for parallel programming," in *IFIP Congress*, 1974, pp. 471–475.

[4] J. B. Dennis, J. B. Fosseen, and J. P. Linderman, "Data flow schemas," in *International Sympoisum on Theoretical Programming*, 1972.

[5] Arvind and R. Nikhil, "Executing a program on the MIT Tagged-Token Dataflow Architecture ," *Computers, IEEE Transactions on*, vol. 39, no. 3, pp. 300 –318, Mar. 1990.

[6] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Computers*, vol. 36, no. 1, pp. 24–35, 1987.

[7] W. Thies, M. Karczmarek, and S. P. Amarasinghe, "Streamit: A language for streaming applications," in *CC*, ser. Lecture Notes in Computer Science, R. N. Horspool, Ed., vol. 2304. Springer, 2002.

[8] G. Berry and L. Cosserat, "The ESTEREL Synchronous Programming Language and its Mathematical Semantics," in *Seminar on Concurrency*, 1984, pp. 389–448.

[9] D. C. Luckham, "Rapide: A language and toolset for causal event modeling of distributed system architectures," in *WWCA*, 1998.

[10] S. A. Edwards and O. Tardieu, "Shim: a deterministic model for heterogeneous embedded systems," *IEEE Trans. VLSI Syst.*, vol. 14, no. 8, pp. 854–867, 2006.

[11] J.-P. Talpin, C. Brunette, T. Gautier, and A. Gamatié, "Polychronous mode automata," in *EMSOFT*, 2006, pp. 83–92.

[12] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Reading, Massachusetts: Addison-Wesley, 1988.

[13] J. C. Hoe and Arvind, "Operation-Centric Hardware Description and Synthesis," *IEEE TRANSACTIONS on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, no. 9, September 2004.

[14] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, 1975.

[15] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," *CWI Quarterly*, vol. 2, pp. 219–246, 1989.

[16] E. Czeck, R. Nanavati, and J. Stoy, "Reliable Design with Multiple Clock Domains," in *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, 2006.

[17] *Bluespec SystemVerilog Version 3.8 Reference Guide*, Bluespec, Inc., Waltham, MA, Nov 2004.

[18] W. Snyder and P. Wasson, and D. Galbi, "Verilator," `http://www.veripool.com/verilator.html`, 2007.

[19] "Carbon Design Systems Inc," `http://carbondesignsystems.com`.

[20] *Catapult-C Manual and C/C++ style guide*, Mentor Graphics, 2004.

[21] Synfora, "PICO Platform," `http://www.synfora.com/`.

[22] AutoESL Desisgn Technologies, Inc., `http://www.autoesl.com`.

[23] A. Agarwal, M. C. Ng, and Arvind, "A comparative evaluation of high-level hardware synthesis using reed-solomon decoder," *Embedded Systems Letters, IEEE*, vol. 2, no. 3, pp. 72 –76, 2010.

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

[25] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A framework for simulating and prototyping heterogenous systems," *Int. Journal in Computer Simulation*, vol. 4, no. 2, pp. 0–, 1994.

[26] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli, "Metropolis: An integrated electronic system design environment," *Computer*, vol. 36, pp. 45–52, 2003.

[27] K. S. Chatha and R. Vemuri, "Magellan: multiway hardware-software partitioning and scheduling for latency minimization of hierarchical control-dataflow task graphs," in *CODES*, 2001, pp. 42–47.

[28] P. Arató, Z. A. Mann, and A. Orbán, "Algorithmic aspects of hardware/software partitioning," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 10, pp. 136–156, January 2005. [Online]. Available: http://doi.acm.org/10.1145/1044111.1044119

[29] R. Ernst, J. Henkel, and T. Benner, "Hardware-software cosynthesis for microcontrollers," *IEEE Des. Test*, vol. 10, no. 4, pp. 64–75, 1993.

[30] N. Dave, Arvind, and M. Pellauer, "Scheduling as Rule Composition," in *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Nice, France, 2007.

[31] N. Dave, M. Pellauer, S. Gerding, and Arvind, "802.11a Transmitter: A Case Study in Microarchitectural Exploration," in *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Napa, CA, 2006.

[32] N. Dave, M. C. Ng, M. Pellauer, and Arvind, "A design flow based on modular refinement," in *Formal Methods and Models for Codesign (MEMOCODE 2010)*.