

Automatic Generation of Hardware/Software Interfaces

Myron King, Nirav Dave, Arvind

Massachusetts Institute of Technology - Computer Science and Artificial Intelligence Laboratory
{mdk,ndave,arvind}@csail.mit.edu

Abstract

Enabling new applications for mobile devices often requires the use of specialized hardware to reduce power consumption. Because of time-to-market pressure, current design methodologies for embedded applications require an early partitioning of the design, allowing the hardware and software to be developed simultaneously, each adhering to a rigid interface contract. This approach is problematic for two reasons: (1) a detailed hardware-software interface is difficult to specify until one is deep into the design process, and (2) it prevents the later migration of functionality across the interface motivated by efficiency concerns or the addition of features. We address this problem using the Bluespec Codesign Language (BCL) which permits the designer to specify the hardware-software partition in the source code, allowing the compiler to synthesize efficient software and hardware along with transactors for communication between the partitions. The movement of functionality across the hardware-software boundary is accomplished by simply specifying a new partitioning, and since the compiler automatically generates the desired interface specifications, it eliminates yet another error-prone design task. In this paper we present BCL, an extension of a commercially available hardware design language (Bluespec SystemVerilog), a new software compiling scheme, and preliminary results generated using our compiler for various hardware-software decompositions of an Ogg Vorbis audio decoder, and a ray-tracing application.

Categories and Subject Descriptors B.6.3 [LOGIC DESIGN]: Hardware description languages; C.3 [SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS]: Real-time and embedded systems; D.3.3 [PROGRAMMING LANGUAGES]: Language Constructs and Features

General Terms Design, Performance

Keywords Hardware/Software Codesign

1. Introduction

Modern mobile devices provide a large and increasing range of functionality, from high-resolution cameras, video and audio decoders, to wireless basebands that can work with a variety of protocols. For power and performance reasons much of this functionality relies on specialized hardware. Often designers start with a pure software (SW) implementation of an algorithm written in C (or Matlab), and identify computationally intensive parts which need to

be implemented in hardware in order to meet the design constraints. Hardware (HW) accelerators come in three different forms. Hardware can be synthesized as an ASIC for the application at hand, in which case the problem reduces to whether the source description is amenable to generation of good quality hardware. The accelerator may also be available in the form of a standardized IP block, like FFT, to be called from the application code. Lastly, the accelerator could be in the form of a programmable processor like a DSP with its own tool chain and distinct programming model. In all three cases, the designer must modify some parts of the application code to make use of the accelerator.

Regardless of what kind of accelerator is eventually used, some software is always required to drive it. 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, and either leave 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, lower-cost or higher-performance alternatives may be ignored.

In our approach, the complete module, as well as the partition boundaries, are specified in the Bluespec Codesign Language (BCL), which is suitable for both hardware and low-level software design. BCL is a semantic extension of Bluespec SystemVerilog (BSV) [1], a commercial language for hardware synthesis. The BCL compiler uses the BSV compiler to generate the special purpose hardware (Verilog) for the hardware partition and it compiles the software partition into C++. It also generates the infrastructure code for communication between the partitions. Crucially, the generated implementations can interoperate with *any* other implementation which conforms to the generated interface. Our approach therefore supports the following design methodologies:

- **Fully Automatic:** The system implementers use the automatically generated implementations directly for both the hardware and software partitions. A major advantage of this approach is that the hardware and software decompositions can be redone with little or no effort to meet changing system requirements as the system design evolves.
- **Partially Automatic:** The system implementers use the automatically generated implementations for either the hardware or software components, choosing some other approach for the remaining parts. While this approach limits the ability to change the partition late in the design process, it does provide implementers with a reference design they can use during testing and development.
- **Interface Only:** The system implementors develop both the hardware and software pieces using alternative approaches, *but use the communication infrastructure code generated by the*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASPLOS'12, March 3–7, 2012, London, England, UK.
Copyright © 2012 ACM 978-1-4503-0759-8/12/03...\$10.00

BCL compiler to define the HW/SW interface. Because the interfaces are backed by fully functional reference implementations, there is no need to build simulators for testing and development purposes.

The primary contributions of this paper are (1) the BCL language, including a method of specifying hardware-software partition at the source level; (2) a compilation strategy for generating efficient code for software partitions; (3) the specification of transactors needed to communicate with the hardware partitions; and (4) a demonstration of the viability of our approach by applying it to concrete examples. 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: In Section 2 we illustrate HW/SW design complications with an example. After a discussion of related work in Section 3, we give an introduction to BCL in Section 4, using Vorbis as a running example. In Section 5 we discuss the operational semantics of the kernel of BCL informally. In Sections 6 we present our compilation strategy, and conclude with a presentation of the experimental results in Section 7.

2. A Motivating Example

Ogg Vorbis is an open-source psychoacoustically-aware audio CODEC aimed at simple low-complexity decoding (see Figure 1). We have chosen this example for its relative simplicity, but the analysis we do applies equally to video CODECs and radio baseband processing [2] which are much more complex.

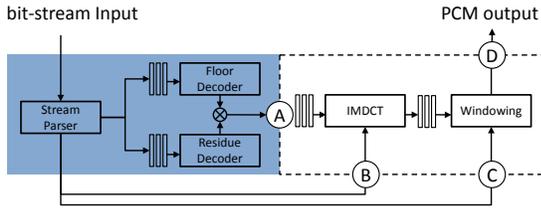


Figure 1. Ogg Vorbis pipeline diagram. The HW accelerator is delimited by the dotted line, and the original SW is shaded

The front end of the pipeline consists of a stream parser and various decoders through which the audio frequency spectra are re-constructed. Due to the compression schema, these modules have complex control, but are relatively lightweight from a computational standpoint. We will assume the front end of the pipeline generates frames, which are then sent to the back-end by invoking the back-end method with a pointer to the frame and the frame size as arguments. The back-end of the pipeline transforms the signal from the frequency to the time domain through the use of an IMDCT which internally uses the computationally intensive IFFT. The frames in the final output are overlapped using a sliding window function to compensate for spectral leakage at the frame boundaries, after which the PCM frames are sent directly to the speaker through memory-mapped IO. The C++ pseudocode for the back-end is shown in Figure 2.

To accelerate this application, it is quite clear that the back-end will require some HW acceleration. However, it is not clear if the entire back-end or just the IFFT should end up in hardware.

2.1 Restructuring Software to Interact with the Accelerator

Let us assume that we are able to implement an efficient IFFT HW accelerator which has an input port and an output port which transfer serialized frames one word at a time. The HW implementation keeps track of the number of words that have been received, and once a full frame is present, it computes the IFFT and subsequently

```
void backend(int* frame, int N){
    int a[2*N], b[2*N], *tmp, i;
    for(i = 0; i < N; i++){
        a[i] = imdctPreLo(i,N,frame[i]);
        a[i+N] = imdctPreHi(i,N,frame[i]);
    }
    swIFFT.ifft(2*N, a, a);
    for(i = 0; i < N; i++){
        b[bitReverse(i)] = imdctPost(i,N,a[i]);
        tmp = swWindow.nextFrame(b,2*N);
    }
    for(i = 0; i < N; i++){
        memcpy(AUDIO_DEV, tmp[i], 4);
    }
}
```

Figure 2. Vorbis back-end in C++

makes the data available on the output port. Let us also assume that the bus protocol permits blocking reads and blocking writes, so if we attempt to read a word from the output port before the answer has been computed, the SW will block. Likewise if we attempt to write a word to the input port before the prior word has been consumed, the write will also block. The invocation of the IFFT in Figure 2 (`swIFFT.ifft(2*N, a, a)`) can be replaced by the code shown in Figure 3 to exploit such HW.

```
...
hwIFFT.setSize(2*N);
for(i = 0; i < 2*N; i++)
    hwIFFT.put(a[i]);
for(i = 0; i < 2*N; i++)
    a[i] = hwIFFT.get();
...
```

Figure 3. Acceleration using blocking HW

Given the latency introduced by transferring the data to and from the accelerator over the bus, it is highly unlikely that the overall algorithm will be sped up by using the accelerator in this manner. In almost all such HW designs where significant data transfer is involved, it is important to find ways to speed up the communication, it is often not possible to change the latency of communication because it is tied to system components like BUS and networks which are typically fixed. Several potential solutions exist to hide this latency:

Communication Granularity: Transferring data one word at a time is inefficient, given the overhead of a bus transaction. This cost can be amortized by transferring larger blocks of data directly into the accelerator memory. At the BUS protocol level, this means exploiting burst transfers. The two loops in Figure 3 which invoke `hwIFFT.put()` and `hwIFFT.get()` respectively will be replaced by two invocations to a burst transfer (see Figure 4). Many systems have dedicated DMA hardware which efficiently implements the desired behavior. Another way to halve the BUS traffic would be to implement the windowing function in the same accelerator and connect the accelerator directly to the speaker.

Pipelining: Often the use of DMA can saturate the BUS bandwidth, and when this happens, the accelerator can become the bottleneck. Pipelining the accelerator hardware can further improve the throughput, and we exploit this concurrency through the use of multithreading, as shown in Figure 4. Though the `putFrame()` and `getFrame()` calls to the HW are still blocking, the pipelined HW implementation can now begin receiving new frames even though it may not have completed the transformation of the previous ones. A separate thread is started at the beginning of time which invokes `backendComplete()` in a `while(true)` loop to drain the HW accelerator and drive the final audio pipeline stages. When running under a traditional OS, this code is actually quite convenient to write, though many embedded operating systems do not support multithreading, requiring the application to simulate this behavior manually.

```

void backend(int* frame, int N){
    int a[2*N], i;
    for(i = 0; i < N; i++){
        a[i] = imdctPreLo(i,N,frame[i]);
        a[i+N] = imdctPreHi(i,N,frame[i]);
    }
    hwIFFT.setSize(2*N);
    hwIFFT.putFrame(a,2*N);
}

void backend_complete(int N){
    int a*, b[2*N], i, *tmp;
    a = hwIFFT.getFrame(2*N);
    for(i = 0; i < N; i++){
        b[bitReverse(i)] = imdctPost(i,N,a[i]);
    }
    tmp = swWindow.nextFrame(b,2*N);
    for(i = 0; i < N; i++){
        memcpy(AUDIO_DEV, tmp[i], 4);
    }
}

```

Figure 4. Multithreaded use of pipelined HW (backend_complete() is repeatedly invoked by a separate thread)

Nonblocking Interface: Suppose that the accelerator is not pipelined, we can still improve the overall efficiency by doing useful work while waiting for the accelerator to complete its work. This requires making the IFFT interface non-blocking, as shown in Figure 5. Nonblocking interfaces require exposing some internal details of the accelerator, e.g., the status of internal state machines or memory buffers. For example, a protocol might require the HW to first indicate that it has enough buffering to receive an entire frame, so that individual puts will never block waiting for space to become available. This information invariably percolates up to the front of the audio pipeline, since we now need to change the interface type of backend to return bool (see Figure 5). Related, but more complicated issues arise when an accelerator is shared among several threads.

```

volatile int* hw_status, *hw_fbuff, *hw_sz;
bool backend(int* frame, int N){
    if(*hw_status == RDY){return false;}
    ...
    *hw_size = (2*N);
    memcpy(*hw_fbuff, a, 2*N*4);
    *hw_status = GO;
    while(*hw_status != RDY){;}
    memcpy(a, *hw_fbuff, 2*N*4);
    ...
    return true;}

```

Figure 5. Acceleration using non-blocking hardware

The software now has much greater control over the hardware, but as a result, the clean get/put interface has now been polluted by implementation detail. Exposing this level of details violates the important design principle of isolation; now if the HW needs to be refined, the SW must be re-validated since it can observe the internal state of the HW.

2.2 Synthesis of the Accelerator

It should be clear from the previous discussions that several different hardware designs – pipelined versus non-pipelined, blocking versus nonblocking – are possible for the IFFT accelerator. Given the state of the art in hardware synthesis, it is not possible to generate efficient hardware from typical software codes written in C or C++. The designer must describe the architecture for the accelerator in some HDL like Verilog. To anyone who is familiar with HDL’s, it should be clear that this is a radical change in the way one thinks about programming. Furthermore, any verification effort performed on the original implementation must be repeated in a new language with completely different semantics, and since the IFFT interface has changed, the SW partition will need to be modified and re-verified as well.

2.3 Data Format Issues

Mismatch of data representation is one of the more common source of errors in HW/SW codesign. The objects generated by the application must be correctly converted to the bit representation expected by the accelerator. This problem is compounded by the fact that the C++ compiler and the Verilog compiler may have completely different layouts for the “same” data structure:

```

C++:
template<typename F, typename I>
struct FixPt{ F fract; I integer; };
template<typename T>
struct Complex{T rel; T img;};

```

```

Verilog:
typedef struct {bit[31:0] fract; bit[31:0] int;} FixPt;
typedef struct {FixPt rel; FixPt img;}; Complex.FixPt;

```

The two languages may have different endian conventions, and even two compilers for the same language can use different conventions depending on the targeted substrate. Many SW representations do not even have a natural HW analog. In general, keeping HW and SW representations of shared data structures is both tedious and error prone.

2.4 A Language-Based Solution

A 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.

The design of BCL was motivated by these goals, and in BCL it is possible to describe any subset of the Vorbis pipeline in such a way that efficient hardware may be synthesized. Different microarchitectures for implementing [say] IFFT can also be described by writing different BCL programs [2], and at the same time, the whole design can be compiled into sequential software. What makes BCL unique is that by using the idea of *computational domains* (Section 4.2), it is possible to specify which part of the BCL program

should be run in HW (*i.e.*, compiled to Verilog) and which in SW (*i.e.*, compiled to C++). Thus, moving any code across the HW/SW boundary is as simple as specifying a different partition, a feature that is especially useful when experimenting with how much of the code surrounding IFFT in the back-end should be implemented as hardware. The use of a single language automatically avoids the data formatting issues mentioned previously, and the code generated by compiling BCL can interface both with other system hardware such as the audio device, and software up the stack starting from the Ogg Vorbis front end.

The computational domains specified in BCL let the compiler automatically deduce the communication between hardware and software. If the accelerator is connected to the microprocessor by a BUS, the compiler will use this underlying substrate through low-level library elements. Note that the interface given for the back-end of the pipeline (behind which all HW/SW codesign exploration takes place) is a much more stable interface than any traditional BUS-based interface used by device drivers.

3. Related Work

We are not aware of any language which satisfies all the goals enumerated in Section 2, however there is a substantial body of work, both academic and commercial, relevant to various aspects of hardware-software codesign.

Implementation-agnostic parallel models: 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 [3] – a popular C++ class library for modeling embedded systems. While these libraries provide great flexibility in specifying modules, the language itself lacks proper compositional semantics, producing unpredictable behaviors when connecting modules. Synthesis of high-quality hardware from SystemC remains a challenge.

Dataflow models, both at macro-levels (Kahn [4]) and fine-grained levels (Dennis [5], Arvind [6]), provide many attractive properties but abstract away important resource-level issues that are necessary 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 [7, 8].

Synchronous dataflow is a clean model of concurrency based on synchronous clocks, and forms the basis of many programming languages (*e.g.*, Esterel [9], Lustre [10], Signal [11], Rapide [12], Shim [13], Polysynchrony [14]). Scade [15], a commercial tool for designing safety-critical applications, is also based on this model.

We have chosen *guarded atomic actions* (or rules) as the basis 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 [16] to describe software, and then by Hoe and Arvind to generate hardware [17]. Dijkstra’s Guarded Commands [18] and Lynch’s IO Automata [19] are also closely related. BCL extends the idea of multiple clock domains [20] from Bluespec SystemVerilog (BSV) [21] to specify how a design should be split between hardware and software. Hardware compilation from BCL is a straightforward translation into BSV, whose subsequent compilation to Verilog is a mature and proven technology [17, 21]. Guarded atomic actions also provide a good foundation to build analysis and verification tools [22].

Generation of software from hardware descriptions: Hardware description languages (HDLs) like Verilog and VHDL provide extremely fine-grain parallelism but lack an understandable semantic model [23] and are also impractical for writing software. Nevertheless, these languages can compile to a software simulator, which

can be viewed as a software implementation. Popular commercial products like Verilator [24] and Carbon [25] 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 performance is often several factors slower than natural software implementations of the same algorithm.

The Chinook [26, 27] compiler addressed an important aspect of the hardware/software interface problem, by automating the task of generating a software interface from the RTL description and the timing information of the hardware blocks.

Lastly, Bluespec’s Bluesim [21] 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.

Generation of hardware from sequential software specifications: To avoid the burden of using low-level HDLs, the idea of extracting a hardware design from a familiar software language, *e.g.*, C, Java, or Haskell, has great appeal. Liao et al. [28] present one of the earliest solutions to mitigate the difficulty of the hardware-software interface problem by describing the interface in C++ and generating an implementation using hand coded libraries that interact with automatically synthesized hardware. Many systems like CatapultC [29], Pico Platform [30], or AutoPilot have been effective at generating some forms of hardware from C code. However, constructing efficient designs with dynamic control can be very hard, if not impossible, using such tools [31].

A related effort is the Liquid Metal project [32] 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.

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 [33] 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 [34], 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 the synthesis of efficient hardware.

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.

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 [35, 36]. 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 is complementary to real hardware-software codesign approaches.

4. Introduction to BCL via Vorbis

BCL is a modern statically-typed language with the flexibility to target either hardware or software. It contains higher-order functions and the rich data structures required to express fine-grain parallelism. Any language with hardware as a target must be restricted so that it is compilable into efficient FSMs. Consequently BCL can be used for writing only the type of software which does not require dynamic heap storage allocation and where the stack depth is known at compile-time. In fact, in a BCL program all state must be declared *explicitly*, and the type system prohibits dynamic allocation of objects. Because BCL's intended use is for describing software primarily at the bottom of the stack, this restriction is not burdensome.

BCL is an extension of Bluespec SystemVerilog (BSV) and borrows BSV's syntax and its powerful Haskell inspired meta-programming features. In the rest of this section we will explain BCL using the Vorbis pipeline and its partitioning between hardware and software.

4.1 The Vorbis Pipeline

Below we have given a simplified version of the code implementing the back-end of the Vorbis pipeline. The reader will notice that the high-level organization is very similar to the code example given in Figure 2:

```

module mkVorbisBackend(VorbisBackend#(k,t))
  IFFT#(2*k, Complex#(t)) ifft <- mkIFFT;
  Window#(2*k, t) window <- mkWindow;
  method Action input(Vector#(k,Complex#(t)) vx)
    Vector#(2*k,Complex#(t)) v;
    for(int i = 0; i < k; i++)
      v[i] = preTable1[i]*vx[i];
      v[K+i] = preTable2[i]*vx[i];
    ifft.input(v);
  rule xfer
    let x = ifft.output() in (ifft.deq());
    Vector#(2*k,t) v;
    for(int i = 0; i < 2*k; i++)
      v[i] = x[bitReverse(i)].real;
    window.input(v);
  rule output
    let rv = window.output();
    window.deq(); AUDIO.DEV.output(rv)

```

A BCL program consists of a hierarchy of modules, and as with any object-oriented language, all interactions with modules occur through their interface methods. The `mkVorbisBackend` module definition instantiates two sub-modules: `ifft` and `window`. Each module is potentially stateful, but ultimately all state is built up from primitive elements called *registers*. It is important not to confuse the state elements with ordinary variables such as “x” or “v”, which are just names. Like functional languages, “x = exp” simply assigns the name x to the expression exp.

Module interfaces are declared separately from the implementation to encourage reuse; the following code defines the polymorphic IFFT interface, parametrized by the type variable t, where t is used to indicate the type of components (e.g., `Fix32`, `Fix16`, `Float63`) used to construct the complex numbers:

```

interface IFFT#(numeric type k, type t)
  method Action input(Vector#(k,Complex#(t)) x)
  method Vector#(k,Complex#(t)) output()
  method Action deq()

```

In BCL, state change is indicated by the `Action` type. The `input` method is of type `Action` and takes as arguments an input frame of size k. This was declared under the assumption that any *implementation* of this interface will store the frame in some internal buffering, changing its state and necessitating the `Action`

type. Methods such as `output` which are not of type `Action` may only read state and compute the return value with a pure function.

The `VorbisBackend` interface (not shown) contains a single action method, `input`. (This interface has no output because the final effect of inputting a frame is the transmission of PCM packets to the audio device.) In contrast to other object-oriented languages, in BCL every method has an associated *guard*, which indicates whether that method is ready to be called. An action method whose guard is false will have no effect, and a value method whose guard is false produces values which cannot be used in further computation; any computation which calls an unready method is itself not valid to be executed. The generation of control logic in the HW implementations is directly based on guard composability.

In our example, the `input` method is only ready to be called once `ifft` is ready to accept the next frame, that is when the internal guard of `ifft.in` is true. We refer to this style of guard as *implicit*, though it is also possible to introduce guards *explicitly* with the **when** keyword. There is no semantic distinction between implicit and explicit guards. Since we cannot determine from outside a module when a particular method will be ready, guarded methods provide a natural abstraction for refining the timing of internal modules.

In addition to the interface methods, the behavior of a module can also be affected by its rules. The module `mkVorbisBackend` contains two such rules, one to transfer data between the `ifft` and `window` modules, and another to transmit PCM packets to the audio device. The invocation of a rule is a *guarded atomic action* and is the only way to modify program state. Just as with action methods, every rule produces a side-effect (state change) which is applied only when the rule's guard evaluates to true. As will become clear in the next section, the rule's guard is the conjunction of all the explicit and implicit guards contained within the constituent actions and expressions.

Rules also provide a natural way to express concurrency. For example the rules `xfer` and `output` may both be executable simultaneously, that is their guards both evaluate to true. Ultimately, all observed behaviors of the rule-based system must be understandable in terms of the execution procedure shown below:

Repeatedly:

1. Choose a rule to execute.
2. Compute the set of state updates and the value of the rule's guard.
3. If the guard is true, apply the updates.

Every rule modifies the state deterministically; non-determinism is introduced through the choice of which rule to fire. This makes a BCL program more like a design specification, and how we resolve this nondeterminism is very important for the quality of the implementation. For example, in hardware we often execute many enabled rules concurrently as long as one-rule-at-a-time semantics are not violated. In sequential SW, on the other hand, we want to compose long sequences of rules in order to exploit data locality. BCL also has the power to let the user specify any level of scheduling constraints [37], but experience shows that designers tend to leave all scheduling decisions to the compiler, except in extremely performance-critical modules.

4.2 Computational Domains

Computational domains, a part of the type system, is the mechanism by which we partition BCL designs between hardware and software. Domains are enforced by annotating every method with a domain name; each rule (or method) can refer to methods in only *one* domain. If a rule refers to methods from domain *D*, we can say that the rule *belongs* to domain *D*, and a simple type invariant can determine if the domain annotations are consistent.

This one-domain-per-rule restriction would seem to preclude inter-domain communication since all data flow occurs through

rules. To enable inter-domain communication, primitive modules called *synchronizers*, which have methods in more than one domain, are provided. Inter-domain communication is possible only through synchronizers, thereby ensuring the absence of inadvertent inter-domain communication (a common pitfall in HW/SW co-design). As we show below, inserting synchronizers *is* the mechanism we provide to specify a program partition. The correct use of synchronizers allows the compiler to automatically infer the domain types on all rules and methods and safely partition a design, while an incorrect use will fail to type check, causing a compilation error.

Suppose we want to implement the IFFT in hardware and the remainder of the Vorbis backend in software. The input and output methods of IFFT must be in the hardware domain (HW) but all other methods must be in the software domain (SW). This can be specified by introducing two synchronizing FIFOs with the following interfaces:

```
interface SyncHtoS#(type t)   interface SyncStoH#(type t)
(HW) Action enq(t val)       (SW) Action enq(t val)
(SW) Action deq()            (HW) Action deq()
(SW) t first()               (HW) t first()
```

These synchronizing FIFOs are inserted in the original Vorbis code, and two new rules are introduced in the HW domain to transmit data to and from the IFFT as shown below:

```
module mkPartitionedVorbisBackEnd(VorbisBackEnd#(t))
IFFT#(Complex#(t)) ifft <- mkIFFT();
Window#(t) window <- mkWindow();
SyncS2H#(Vector#(2*MAX, Complex#(t)))inSync<-mkSyncS2H;
SyncH2S#(Vector#(2*MAX, Complex#(t)))outSync<-mkSyncH2S;
method Action input(vx)
Vector#(2*K,Complex#(t)) v;
for(int i = 0; i < K; i++)
v[i] = preTable1[i]*vx[i];
v[K+i] = preTable2[i]*vx[i];
inSync.in(2*K,v)
rule feedIFFT
let rv = inSync.first(); inSync.deq();
ifft.input(rv);
rule drainIFFT
let rv = ifft.output(); ifft.deq();
outSync.enq(rv);
rule xfer
let x = outSync.first(); outSync.deq();
Vector#(2*K,t) v;
for(int i = 0; i < 2*K; i++)
v[i] = x[bitReverse(i)].real;
window.input(v);
rule output
let rv = window.output(); window.deq();
AUDIO_DEV.output(rv);
```

Though moving the design from a single domain to multiple domains required modifying only a few lines of code, we have introduced buffering along the HW/SW cut. In general such buffering can change the behavior of the design. However if we restrict such changes to interfaces that are *latency-insensitive*, then such changes are correct by construction. This property of interfaces also enables modular refinement of a design; moving a module from hardware to software *is* modular refinement [38]. A method to automatically verify these types of refinements is discussed by Dave et al [22].

Domain Polymorphism: BCL also allows domain names to be passed as (type) parameters. This permits us to write very general partitioned codes where a domain can be moved from software to hardware or vice versa without changing the source code. We illustrate this next.

```
interface Sync#(type t, domain a, domain b)
(a) Action enq(t val)
(b) t first()
(b) Action deq()
```

We can then declare the internal synchronizers in the following manner, where **a** is a free type variable:

```
Sync#(Vector#(2*MAX, Complex#(t)), a, HW)
```

```
inSync <- mkSync();
Sync#(Vector#(2*MAX, Complex#(t)), HW, a)
outSync <- mkSync();
```

The resulting Vorbis module is fully polymorphic in its domain type. If the parameter **a** is instantiated to HW, the compiler will replace the synchronizers with lightweight FIFOs since no actual synchronization is necessary. If the domain type variable **a** is instantiated as “SW” then the synchronizers would become mkSyncHtoS and mkSyncStoH respectively. In other words, a very general partitioned code may insert more synchronizers than necessary for a specific partition, but these can be optimized by the compiler in a straightforward manner.

4.3 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 once type checking has completed. As shown in Figure 6, the hardware partition of the mkPartitionedVorbisBackEnd module will contain the rules *feedIFFT*, and *drainIFFT*, and the software partition will contain the rest. Once separated, *each partition can now be treated as an distinct BCL program, which communicates with other partitions using synchronizer primitives*. The overall semantics of the original (un-partitioned) program will be preserved, since the synchronizers enforce the property of *latency-insensitivity*. We refer to the data-flow graphs describing such partitioned programs as LIBDNs (Latency-Insensitive Bounded Dataflow Graphs)[39]. If used correctly, the Sync FIFOs in the BCL programs are equivalent to LIBDN FIFOs.

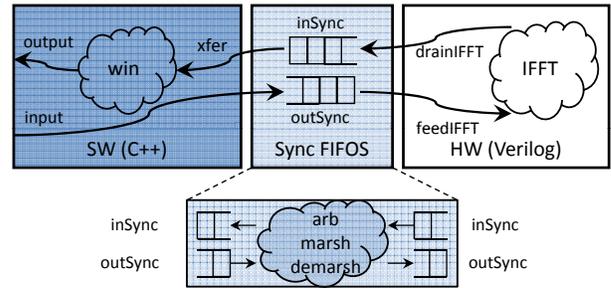


Figure 6. Each BCL program compiles into three partitions: HW, SW, and Interface. Each synchronizer is “split” between hardware and software, and arbitration, marshaling, and de-marshaling logic is generated to connect the two over the physical channel.

4.4 Mapping Synchronizers

In addition to the generation of the HW and SW partitions as shown in Figure 6, the compiler must generate code to connect the two on a particular platform. A partitioned code often requires more synchronizers than the number of physical channels available in a system (typically *one*, in the form of a bus). In addition to arbitration of the physical communication substrate among multiple LIBDN FIFOs, the compiler handles the problem of marshaling and de-marshaling messages. For example, in our program we specified atomic transfers at the audio-frame granularity, but the transmission of these “messages” occurs at a granularity which is dictated by the physical substrate. Through analysis, the compiler is sometimes able to transform the marshaling and demarshaling loops into more efficient burst transfers.

The low-level details of bus transactions or on-chip networks are tedious and ever-changing. For each “supported” platform, these must be manually abstracted as simple get/put interfaces. Using such interfaces, the BCL compiler synthesizes the appropriate hardware and software to map the LIBDN FIFOs onto the available

physical channels. With the use of sufficient buffering and virtual channels representing each LIBDN FIFO, we can guarantee that no new deadlocks or head-of-line blocking will be introduced. Our compiler generates the necessary software and hardware infrastructures, assuming that the HW partition fits on a single FPGA. Fleming et al. have synthesized a similar communication infrastructure for multiple FPGA platforms [40]. It is possible to generalize either scheme to handle multiple computational domains for arbitrary physical topologies.

4.5 Implementing the IFFT Module in BCL

Our partitioned example now assumes that the IFFT functionality will run in hardware, but we have not yet shown how BCL can be used to design efficient HW. There is substantial literature on this topic, but here we provide a short discussion in terms of IFFT which also illustrates the tradeoffs between hardware and software quality. Though the IFFT interface was specified to operate on dynamic sizes, the resulting mechanism needlessly obscures the discussion. Therefore, we present the subsequent designs assuming a fixed frame size of sixty-four, though the reasoning applies equally to the fully flexible design.

Unpipelined: Perhaps the most natural description of the IFFT is the nested `for` loop contained within a single rule, shown below:

```

module mkIFFTComb (IFFT#(k, t))
  FIFO#(Vector#(k, t)) inQ <- mkFIFO()
  FIFO#(Vector#(k, t)) outQ <- mkFIFO()
  method Action input (Vector#(k, t) x)
    inQ.enq(x)
  rule doIFFT
    let x = inQ.first(); inQ.deq();
    for (int stage = 0; stage < 3; stage++)
      for (int pos = 0; pos < 16; pos++)
        x = applyRadix(stage, pos, x);
    outQ.enq(x);
  method Vector#(k, t) output()
    return outQ.first();
  method Action deq()
    outQ.deq();

```

In this module, all the computation is done by the `doIFFT` rule, but to correctly implement the asynchronous interface, two FIFOs are used to buffer values. In software, the rule will compile into efficient loops, while in hardware they will be fully unrolled into a single block of combinational logic. Anyone with experience in hardware design will recognize that this circuit will produce an extremely long combinational path which will need to be clocked very slowly.

Pipelined: A better implementation would compute each stage of the IFFT in separate cycles. This cuts the critical path in hardware and introduces interesting pipelining opportunities. To do this we need to add some extra buffering in the form of additional FIFOs:

```

module mkIFFTPipe (IFFT#(k, t))
  Vector#(4, FIFO#(Vector#(k, t))) buff <- replM(mkFIFO);
  method Action input (Vector#(k, t) x)
    buff[0].enq(x);
  for (stage = 0; stage < 3; stage++) // gen. the rules
    rule stageN
      let x = buff[stage].first(); buff[stage].deq();
      for (int pos = 0; pos < 16; pos++)
        x = applyRadix(0, pos, x);
      buff[stage+1].enq(x);
  method Vector#(k, t) output()
    return buff[3].first();
  method Action deq()
    buff[3].deq();

```

The `for`-loop which encloses the rule `stageN` is meta-syntax for writing three rules manually. These rules are generated by unfolding the loop. Consequently, in this module each stage is represented by a single rule which can fire independently. The implicit conditions (formed by the internal conditions on the FIFOs) in each stage

rule enforces the necessary data dependencies to guarantee correct results are generated.

When compiling `mkIFFTPipe` to software we want to exploit data locality and process a single frame at a time, a task which can be accomplished by executing each IFFT stage in data-flow order. This reconstruction of the original outer-level loop and has the effect of “passing the algorithm over the data”. In hardware, on the other hand, we want to exploit pipeline parallelism and in each clock cycle run each rule once on different data, an approach which can be characterized as “passing the data through the algorithm”. There are a host of other variations on this pipeline which are easily specified in BCL which we do not discuss for the sake of brevity. It should be noted that as the implementations become more “hardware-friendly” the task of synthesizing efficient software often becomes increasingly challenging.

5. Kernel of BCL

The kernel of BCL that we describe in this section consists of the language once type checking has been performed, all modules have been instantiated, and all meta-linguistic features have been eliminated by a compiler phase known as *static elaboration*. We explain the main features of kernel BCL intuitively; an SOS-style semantics of kernel BCL can be found in [37].

BCL Grammar: The grammar for BCL is given in Figure 7. A BCL *module* consists of 3 major parts: a set of state element variables (*i.e.*, registers or other module instantiations), a set of guarded atomic actions or *rules* which describe the internal state changes, and a set of *methods* which implement the interface through which the rest of the design interacts with this module. The Module definition includes a constructor argument list $[t]$, and each module instantiation *Inst* contains a list of values $[v]$ to be passed to the module constructor when building a module hierarchy. A BCL *program* consists of a list of module definitions with one module specified as the root or the main program. Instantiating the root recursively instantiates the entire program state.

$m ::=$	Module mn $[t]$	// Name, Argument list
	[Inst mn n $[c]$	// 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
	$a \mid a$	// Parallel composition
	$a ; a$	// Sequential composition
	a when e	// Guarded action
	$(t = e$ in $a)$	// Let action
	loop e a	// Loop action
	localGuard a	// Local guard action
	$m.g(e)$	// Action method call 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 method call f of m
$op ::=$	&& ...	// Primitive operations
$pr ::=$	$[m]$ $(mn, [c])$	// BTRS program

Figure 7. BCL Grammar

Rules: A rule in BCL is a guarded atomic action, and whenever a rule is evaluated it yields a Boolean guard value and a set of state (register) updates. If the guard is true then the system is allowed

A.1	$(a_1 \text{ when } p) \mid a_2$	\equiv	$(a_1 \mid a_2) \text{ when } p$
A.2	$a_1 \mid (a_2 \text{ when } p)$	\equiv	$(a_1 \mid a_2) \text{ when } p$
A.3	$(a_1 \text{ when } p) ; a_2$	\equiv	$(a_1 ; a_2) \text{ when } p$
A.4	if $(e \text{ when } p)$ then a	\equiv	(if e then $a)$ when p
A.5	if e then $(a \text{ when } p)$	\equiv	(if e then $a)$ when $(p \vee \neg e)$
A.6	$(a \text{ when } p) \text{ when } q$	\equiv	$a \text{ when } (p \wedge q)$
A.7	$r := (e \text{ when } p)$	\equiv	$(r := e) \text{ when } p$
A.8	$m.h(e \text{ when } p)$	\equiv	$m.h(e) \text{ when } p$
A.9	Rule n if p then a	\equiv	Rule n $(a \text{ when } p)$

Figure 8. When-Related Axioms on Actions

to make a transition to the new state, *i.e.*, update the state. If the guard is false then the rule has no effect. As discussed in Section 4, a scheduler selects a rule to execute. Since several different rules can have their guards true in a given state, the choice of which rule to execute can lead to different execution paths. This source of non-determinism is an important aspect of BCL.

Actions: The primitive action in BCL is the assignment of a value to a register, written as “ $r := e$ ”. Since expressions are guarded, it is possible that in a given state, e ’s guard may be false, that is, e is not ready to be evaluated. By the axioms described in Figure 8, the guard on entire assignment action is false, and the value stored in r is not updated.

Two actions can be composed either in *parallel* or in *sequence* to form a bigger atomic action. If $A_1 \mid A_2$ are composed in parallel both observe the same initial state and do not observe each other’s updates. Thus the action $r_1 := r_2 \mid r_2 := r_1$ swaps the values in registers r_1 and r_2 . If both A_1 and A_2 try to update the same register, the parallel action is illegal and generates a DOUBLE_WRITE_ERROR. Sequential composition is more in line with other languages with atomic actions. The action $A_1 ; A_2$ represents the execution of A_1 followed by A_2 . A_2 observes the full effect of A_1 . No other action observes A_1 ’s updates without also observing A_2 ’s updates. In both cases, a guard failure in the evaluation of either action invalidates the composed action.

Conditional versus Guarded Actions: BCL has both conditional actions (**ifs**) as well as guarded actions (**whens**). These are similar as they both restrict the evaluation of an action based on some condition. The difference is their scope of effect: conditional actions have only a local effect whereas guarded actions have a global effect. If an **if**’s predicate evaluates to false, then that action doesn’t happen (produces no updates). If a **when**’s predicate is false, the subaction (and as a result the whole atomic action that contains it) is invalid. One of the best ways to understand the differences between **ifs** and **whens** is to examine the axioms in Figure 8.

Axioms A.1 and A.2 collectively state that a guard on one action in a parallel composition affects all the other actions. Axiom A.3 deals with a particular sequential composition. Axioms A.4 and A.5 state that guards in conditional actions are reflected only when the condition is true, but guards in the predicate of a condition are always evaluated. A.6 deals with merging **when** clauses. A.7 and A.8 translate expression **when**-clauses to action **when**-clauses. Axiom A.9 states that top-level **whens** in a rule can be treated as an **if** and vice versa.

`localGuard(a)`, instead of propagating a guard failure in action a to the enclosing rule, effectively converts a to `noAction`. This gives the programmer the power to indirectly predicate behavior upon implicit guard failures and can be used to construct a dynamic-length *sequence* of atomic actions. For example, we might want to implement a “non-atomic atomic loop”, in which the loop body itself is atomic, but loop termination is triggered by the final guard failure as shown below:

```
cond.reg := true;
loop (cond.reg = true)
  cond.reg := false;
  localGuard {<loopbody> | cond.reg := true}
```

Expressions: The only thing special about expressions in BCL is guards. However guards in expressions can be lifted to the top expression using axioms which are similar to the when-axioms for actions in Figure 8. After **when** lifting, we end up with “ $e \text{ when } g$ ” where both e and g are free of guards.

Module boundaries represent a slight complication in lifting guards. It is best to think of every method call $f(e)$ as a pair of unguarded method calls $f_b(e)$ and $f_g(e)$ containing the method body and guard respectively. Any invocation of $f(e)$ can be replaced with the expression “ $f_b(e) \text{ when } f_g(e)$ ”.

Strict Method Calls and Non-Strict Lets: BCL has non-strict (sometimes called lazy) lets and procedure calls. Consequently spurious let bindings have no effect and we have more powerful algebraic laws for program transformation. However, BCL has strict method calls because each method represents a concrete resource, a *port*, in our implementation. Thus we expect the semantics of the program to be affected if a module is merged with the surrounding module.

6. Implementing BCL

We have implemented a compiler which generates a C++ implementation of the software partition of a BCL specification. This is then compiled, along with some libraries, into an executable program. For hardware partitions, our compiler generates BSV, which is compiled into Verilog using BSC from Bluespec Inc. We discuss software generation at length but describe hardware generation only briefly because it has been published elsewhere.

6.1 Shadowing State

At a high level, the software implementation of BCL can be explained by pointing out the differences between rule-based systems and Transactional Memory. Both these systems have the *linearizability* property, that is, the semantics of the system are understandable by executing rules (transactions) in some sequential order. However, the implementation techniques for the two systems are quite different. In TM systems, multiple transactions are executed eagerly, and all except one are aborted if a conflict is detected. Upon failure, TM systems transfer control back to the transaction’s boundary, and the transaction is re-executed. In TM systems, conflicts are typically defined in terms of read/write conflicts, though more sophisticated schemes exist. The challenge in the implementation of TM systems is to find low-cost mechanisms for detecting conflicting transactions and rolling back the computation in case of an aborted transaction. The possibility of a rollback invariably requires shadow states or check-points.

The implementation of rule-based systems such as BSV and BCL, on the other hand, pre-schedule rules in such a way that conflicting rules are never executed simultaneously. Thus, there is no need for dynamic conflict detection. Typically, many rule guards are evaluated in parallel, and among the rules whose guards are *true*, all non-conflicting rules are scheduled simultaneously. The compiler does pair-wise static analysis to conservatively estimate conflicts between rules. Thus, compilers for rule-based systems construct a scheduler for each design using the static conflicts, as well as dynamic information about which rules are ready to *fire*.

Even though we never simultaneously schedule conflicting rules, our implementation still requires shadow states. In simple designs all guards can be lifted out of the rule, and the scheduler executes a rule only when its guard is true. In complex rules, not all guards can be lifted, which leaves the possibility of internal guard failure. It is this possibility of failure which necessitates shadows in our system.

Parallel Composition: There is a complication which arises from the fact that GAAs might contain internal concurrency (due to the use of parallel action composition). In general, “ $A_1 \mid A_2$ ” will

require creating two shadows of the incoming state to execute A1 and A2 in isolation. After the execution of A1 and A2, the copies will have to be merged into a single shadow. This merge may produce a dynamic error if A1 and A2 modify the same state element; such actions are illegal. The following example illustrates this, where $c1$ and $c2$ are dynamic boolean expressions, f is a fifo, and a and b are registers:

```
(if (c1) then a := f.first(); f.deq()) |
(if (c2) then b := f.first(); f.deq())
```

If $c1$ and $c2$ both evaluate to true and f is not empty then both the sub-actions will attempt to dequeue the same element, which is an error. Even if there is no danger of parallel merge errors, shadow state is required, as illustrated by the following simple example, which swaps the values of two registers: “ $a := b \mid b := a$ ”.

6.2 Control Transfer

The C++ implementation of BCL programs can be described informally as follows: Each BCL module is implemented as a C++ class definition. The user-specified rules and methods are implemented as class member functions, and the sub-modules are implemented as member variables. Each class definition is augmented with a number of additional member functions and variables, which together constitute light-weight transactional run-time, and execution strategy.

A copy of the original state is allocated at program start-up time, and is referred to as the *persistent* shadow. This shadow is populated (in a change-log manner) as the rule executes. When a rule executes to completion, the shadow state is committed to the original program state. Consistency with the original state is maintained by executing a rollback if a rule fails. Unlike the shadows for rule execution, those required for parallel action execution are allocated and de-allocated dynamically. We chose this approach because after optimizations, parallel compositions are relatively infrequent.

Because of the need for compositional method invocations, guard failure is implemented using the C++ `throw` command, which is always caught by the rule in which the action is being executed. Hence, the structure of rule bodies is always a try/catch block, with commit routines at the end of the try block and the necessary rollback code in the catch block. The *localGuard* construct also requires dynamic shadow creation and because it doesn't propagate rule failures, the C++ implementation of this construct also uses try/catch blocks.

6.3 Code Transformation: Reducing the Cost

In this section, we discuss some simple transformations which can drastically improve the code quality.

Lifting Guards: When executing a rule whose guard might fail, it is preferable that it happen as early in the execution as possible, since early failure avoids the useless execution of the remainder of the rule body. Consider the following transformation of a BCL expression:

$$(A1 \text{ when } A1_g \mid A2 \text{ when } A2_g) \Rightarrow (A1 \mid A2) \text{ when } (A1_g \wedge A2_g)$$

By rules given in [37], it is possible to lift most guards to the top level. In some cases, we can transform rules to the form “A when E” where A and E are themselves guard-free. In this form, we can guarantee that if E evaluates to ‘True’, A will execute without any guard failures. With this knowledge, we can avoid the cost of using a try/catch block to detect guard failures in sub-module method invocations, and also perform the computation *in situ* to avoid the cost of commit entirely. The when-axioms (Figure 8) state that guards cannot be lifted through the sequential composition of actions or loops, which is why we cannot do away with state shadowing all together.

Avoiding Try/Catch: Even when guards cannot be lifted completely, we can still improve the quality of code when all methods in a rule are inlined, or at least determine through static analysis that those remaining (methods on primitive modules) will not fail. In this case, we can dispense with the top-level try/catch block and instead handle all constituent **when** clauses explicitly, branching directly to the rollback code. Try/catch blocks in C++ are not cheap, and this analysis proves to be quite effective. To illustrate this optimization, consider the BCL rule “**Rule** `foo {a := 1; f.enq(a); a := 0}`”. The code generated without method inlining is give in Figure 9, while the (far more efficient) code generated after method inlining is given in Figure 10.

```
void foo(){
  try{
    a.s.write(0);
    f.s.enq(a.s.read());
    a.s.write(1);
    f.commit(f.s);
    a.commit(a.s);
  }catch{
    a.s.rollback(a);
    f.s.rollback(f);
  }
}
```

Figure 9. without inlining

```
void foo(){
  a.s.write(0);
  if(f.s.can_enq()){
    f.s.enq(a.s.read());
  }else
    goto rollback;
  a.s.write(1);
  f.commit(f.s);
  a.commit(a.s);
rollback:
  a.s.rollback(a);
}
```

Figure 10. with inlining

Sequentialization of Parallel Actions: The cost of shadowing for parallel actions can be reduced by transforming them into an equivalent sequential composition of actions. For example, $(A \mid B)$ is equivalent to $(A ; B)$ if the intersection of the write-set of action A and the read-set of action B is empty. The equivalence of $(A \mid B)$ and $(B \mid A)$ gives us even more flexibility in finding an equivalent sequential form. In addition, action methods invoked on the same module but in parallel branches can be invoked in sequence if they modify disjoint internal states. In both cases, since the two actions were originally composed in parallel, their respective guards may be lifted, giving us a sequential composition *and* the ability to guarantee that the second action will complete without a guard failure. Since parallel composition is most naturally expressed in hardware and sequential composition in software, recognizing these patterns is important when implementing a “hardware” module in software. Sometimes sequentialization of parallel actions is made possible by introducing some shadow state (consider the swap example). Even this turns out to be a win because static allocation of state is more efficient than *dynamic* allocation.

Partial Shadowing: When we must create shadow state for an action, we need not shadow the entire module: only those state elements which can *actually* be modified by the action need to be shadowed. This selective shadowing provides substantial savings in many cases, and opportunities for these savings increase if we inline methods or perform some static analysis. Loops and localGuard provide additional opportunities for partial shadowing.

Scheduling: In both hardware and software, the execution strategy has a great impact on performance. The most important concern in scheduling software is to choose a rule which will not fail, since partial execution of any rule is wasted work (the cost of partial execution and rollback). We are investigating the use of user-provided scheduling annotations to improve compiler analysis, but do not discuss it in this paper.

Parts of programs are sometimes identifiable as synchronous regions in which case well-known static scheduling techniques are applied [7]. When available, these synchronous guarantees can lead to very efficient code generation in both hardware and software implementations. Even without such synchronous guarantees, the compiler can exploit data-flow analysis which may reveal that the

execution of one rule may enable another, permitting the construction of longer sequences of rule invocations which successfully execute without guard failures, an important element in efficient software. This strategy is most effective when the program data-flow graph is sparse.

To illustrate the scheduling challenge, consider the following rule, which transfers a single audio frame between a producer module (p) and consumer module (c) non-atomically. If p empties or c is full before the complete frame has been transferred, the scheduler will attempt to complete the transfer during a subsequent rule invocation:

```
rule xferSW when(True)
  cond.reg := true;
  loop (cond.reg = true && cnt < frameSz)
    cond.reg := false;
    localGuard
      cond.reg := true ; cnt := cnt+1;
      let w = p.output(); p.deq(); c.input(w)
```

The sequential composition inherent in loops is not directly implementable in HW. For a hardware implementation, we might instead use a rule which transmits one word at a time, and rely on the scheduler to attempt to execute it once every HW clock cycle. The effects of the resulting non-atomic transfer of a single frame is identical, though the schedules are completely different:

```
rule xferHW when(cnt < frameSz)
  let w = p.output(); p.deq();
  c.input(w); cnt := cnt+1
```

Since one of our goals is to write code which can be moved between HW and SW with minimal changes, we need some way of resolving these two very different idioms. In essence, we would like to transfer a frame from p to c using `xferHW` with the same efficiency as if we had used `xferSW`. If the SW scheduler invokes `xferHW` in a loop, the overall performance of the transfer will not suffer. If the rules in c require a complete frame before they are enabled, any attempts to execute those rules by the scheduler before `xferHW` has been invoked ‘frameSz’ times will be wasted effort. By employing completely different schedules, we are able to generate both efficient HW and SW from the same rules.

Finally, the software implementation presented in this paper is single-threaded: an obvious scheduling improvement would be to use multithreading to exploit rule-level parallelism. Work on this front is ongoing, but beyond the scope of this paper.

6.4 HW Generation

Techniques for hardware generation from rules are well understood and used successfully in the BSV compiler. The initial implementation of this tool was done by Lennart Augustsson, and was then commercialized by Bluespec Inc. With the exception of loops and sequential composition, BCL can be translated to legal BSV, which is then compiled to Verilog using the BSV compiler. We give a short sketch of the BSV compiler, and leave the reader to follow the references for further details.

To implement a rule directly in hardware, we require some notion of shadow state so that we can unwind a computation if we encounter a guard failure in the evaluation. These shadows can require a substantial hardware cost if they are implemented via stateful constructs. However, if we store them ephemerally in wires, they become cheap. The guard drives a multiplexer on the registers holding the module state, which updates them only if the guard evaluates to true. In synchronous hardware, we can guarantee that all shadows are implementable in wires as long as each rule is executed in a single clock cycle. This is the primary optimization which enables the compilation of guarded atomic actions into efficient synchronous circuits [17, 41, 42].

Since, loops with dynamic bounds can’t be executed in a single cycle, such loops are not directly supported in BSV. A technique

has been proposed to introduce rules which can be executed over multiple cycles [43] which provides a solution to this limitation.

7. Experiment and Results

The compiler we have implemented takes a BCL program as input, partitions the design based on the use of synchronizer modules, and outputs the following three components: (1) a C++ implementation of the SW partition, complete with a concrete rule schedule and a driver, (2) a BSV implementation of the HW partition, and (3) infrastructural code to connect the two partitions on supported platforms.

Mobile devices use ASIC implementations for HW acceleration, and our compiler generates Verilog from which such ASIC implementations can be synthesized. Since we do not synthesize ASICs, HW/SW co-simulation is required to test our designs. This type of simulation is very slow, so instead we have chosen an FPGA platform on which to demonstrate our techniques. The conclusions about the quality of the partitions are slightly skewed by the inferior performance of an FPGA when compared to ASIC, but the goal of this paper is not *necessarily* to determine the best partition for these examples, but rather to demonstrate a technique. In spite of this, the *relative* performance of modules running on the FPGA versus the microprocessor gives us valuable insight.

We explored six different HW/SW partitions of the Vorbis decoder, as well as four different partitions of a realistic ray-tracer, both implemented entirely in BCL. The performance numbers were generated in the *fully automatic* mode to demonstrate the efficiency of all components. We show that not only does the relative performance of the various decompositions match our intuition, but that the absolute performance of the full-software partition is competitive with hand-written software. The performance of the generated hardware has already been shown to be as good as hand-written versions [2]. The quality of these results demonstrate the suitability of BCL for all the use cases discussed in Section 1.

Experimental Platform: Our experimental setup is shown in Figure 11. The synchronizer modules operate on the PCI Express bus as well as the internal Xilinx bus connecting the PPC440 to the FPGA, and we ran the experiments in both contexts. Since the performance characteristics of the embedded configuration are more closely aligned with mobile platforms, we use it to generate all the results reported in this section.

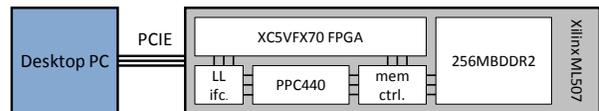


Figure 11. Experimental Setup

The software running on the microprocessor communicates directly to the hardware running on the FPGA through the Local Interconnect™ interface using embedded HDMA engines. The synchronizers themselves form a thin layer on top of this link to provide a FIFO abstraction. Through the synchronizers, we achieve a round-trip latency of approximately 100 FPGA cycles, and are able to stream up to 400 megabytes per second from DDR2 memory to the FPGA modules. For these tests, we clock the PPC440 at 400 Mhz and the FPGA at 100 Mhz.

7.1 Ogg Vorbis

The different HW/SW partitions are shown in Figure 12. The initial stages of the Vorbis pipelined were written manually in C++, and the back-end was specified in BCL. Once the code was written and tested, the specification of the partitions and generation of results required only a few hours.

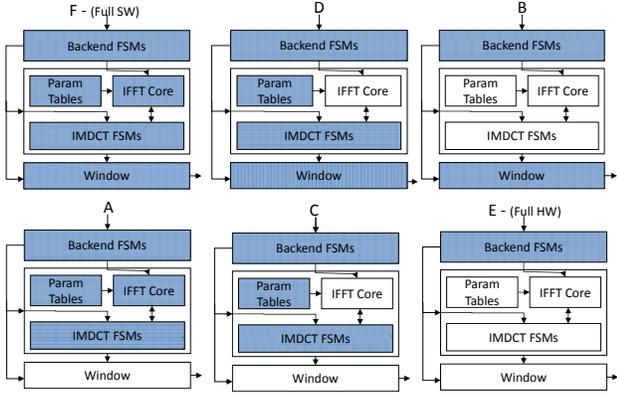


Figure 12. Ogg Vorbis partitions. (SW modules are shaded). The output from the windowing function is always in SW (not shown)

We constructed a test bench consisting of 10000 Vorbis audio frames, and all computation was done using 32-bit fixed point values with 24-bits of fractional precision. The performance results are shown in Figure 13. This graph shows us that moving computation to the hardware does not always speed up the execution; the slowest partition is *not* the one which computes everything in SW (F). In fact, partitions A and C are both slightly slower than F. We know that each link between the HW and SW partitions has a cost, and in order for the net speedup from moving a module from SW to HW to be positive, the speedup observed in the module itself must outweigh the cost of the communication. The performance numbers show that moving the windowing function to HW is not worth the communication overhead. Additionally, notice that the effect of moving *only* the IFFT to HW is marginal. This is because IMDCT FSMs invoke IFFT repeatedly to compute a single output, and transferring this quantity of data between HW and SW is costly. It should be noted that if the audio output could be driven directly from a HW partition, the implementation of windowing in HW would *always* improve performance. These are by no means unexpected results; the novelty is that we are able to generate these numbers with such ease.

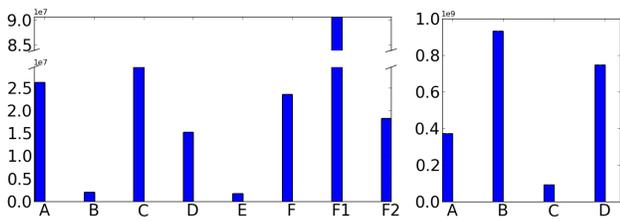


Figure 13. Execution times of OggVorbis (left) and RayTrace (right) partitions listed in FPGA cycles. SW implementations F1 and F2 are hand-coded SystemC and C++ respectively

To compare the performance of our language to more conventional methods of HW/SW codesign, we implemented the pure software partition (F) in SystemC and manually in C++. We chose SystemC to establish an upper bound since it is widely used in HW/SW codesign (though generally in the *modeling* of systems), and the performance is considered by some to be realistic enough to drive design decisions. We chose manual C++ as a lower bound, since this is how embedded devices are commonly written. The SystemC implementation is roughly 3x slower due to the required overhead of modeling all the simulation events. The manual C++ version is slightly faster than the generated one, as it avoids all discarded work or need for shadow state.

7.2 Ray Tracing

A realistic ray tracer was chosen as an additional application with which to evaluate the use of BCL. Since scene geometries can be quite large, memory bandwidth is often as important a concern as the actual compute required to detect ray/geometry intersections. Figure 14 shows the microarchitectural detail, as well as the four HW/SW decompositions considered in this experiment

Once the geometry has been loaded to memory, the module labeled “BVH Ctor” performs an initial pass to construct a bounding volume hierarchy (BVH). With the scene in this form, we can perform $\log(n)$ intersection tests instead of n in the number of scene primitives. The module labeled “Ray Gen” constitutes the first step of the pipeline, computing rays which originate at the camera position and pass through each point in the field of view at a specified granularity. The module labeled “BVH Trav” implements the core of the ray tracing pipeline. As the algorithm traces a ray through the scene, it traverses the BVH by repeatedly testing for intersection with the sub-volumes, and finally with the geometry primitives at the leaves of the tree. After a fixed number of reflections/refractions are calculated, the final color is stored in a bitmap. Unlike the Vorbis example, which is amenable to pipeline parallelism, efficient ray tracing requires us to exploit data parallelism.

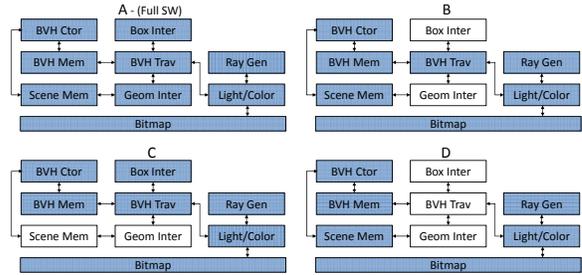


Figure 14. Ray tracing partitions with SW modules shaded

The performance numbers for ray tracing (Figure 13) were generated using a small benchmark consisting of 1024 geometry primitives. This application provides a multitude of different partitioning options, and we chose four which illustrate important properties of the program’s data flow. The fastest partitioning given (C) has the ray/geometry intersection engine implemented in hardware, and the scene geometry stored in low-latency-access on-chip block RAMs. Collision testing requires a substantial amount of arithmetic, which we are able to accelerate through aggressive pipelining. Of course, if the number of geometry primitives falls below some threshold, a full SW implementation might be faster. Configurations B and D, though they both use HW acceleration, are slower than the pure software implementation because the savings in computation are outweighed by the incurred cost of communication.

8. Conclusion

In this paper we have explored a new method for HW/SW codesign, where instead of requiring precise interface specification a priori, the user specifies the entire design, *including* the HW/SW decomposition, in a single language. The resulting specification can be used to generate precise implementations which encapsulate low-level properties of the communication substrate inside the synchronizer modules. We have implemented a complete system, including a compiler and run-time, and have illustrated our approach using examples written in BCL. We have shown that partitioning designs in BCL is natural and easy and that the reduced effort of partitioning allows the programmer to concentrate exclusively on the performance implications of the partitioning choices. BCL concedes nothing in the quality of hardware or software that is generated. The same exercise using conventional multi-language methodologies would require far greater effort.

Future work includes applying BCL to exploit multiple software threads and multiple hardware substrates, the improvement of generated software through more complex analysis, the investigation of additional HW/SW codesign platforms for *exploring HW/SW tradeoffs*, and gaining more insight into which BCL patterns are best suited for generating both efficient HW and SW.

Acknowledgments

This work was supported by the NSF (#CCF-0541164), and a NRF grant from the Korean Government (MEST) (#R33-10095). The authors thank Jamey Hicks and John Ankorn of NRC-Cambridge for their input at the start of this project, the team at Bluespec Inc. for their technical support, Asif Khan of CSAIL for his invaluable help with FPGA platforms, and Martin Rinard of CSAIL for his input in the writing process. Discussions with fellow CSG members and Professor Jihong Kim of Seoul National University have also improved this work.

References

- [1] L. Augustsson, J. Schwarz, and R. S. Nikhil, "Bluespec Language definition," p. 95, 2001, Sandburst Corp.
- [2] M. C. Ng, M. Vijayaraghavan, G. Raghavan, N. Dave, J. Hicks, and Arvind, "From WiFi to WiMAX: Techniques for IP Reuse Across Different OFDM Protocols," in *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Nice, France, 2007.
- [3] 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.
- [4] G. Kahn, "The semantics of simple language for parallel programming," in *IFIP Congress*, 1974, pp. 471–475.
- [5] J. B. Dennis, J. B. Fosseen, and J. P. Linderman, "Data flow schemas," in *International Symposium on Theoretical Programming*, 1972.
- [6] 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.
- [7] 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.
- [8] 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.
- [9] G. Berry and L. Cosserat, "The ESTEREL Synchronous Programming Language and its Mathematical Semantics," in *Seminar on Concurrency*, 1984, pp. 389–448.
- [10] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud, "The synchronous data flow programming language lustre," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1305–1320, sep 1991.
- [11] J.-L. Colaço, G. Hamon, and M. Pouzet, "Mixing signals and modes in synchronous data-flow systems," in *Proceedings of the 6th ACM & IEEE International conference on Embedded software*, ser. EMSOFT '06. New York, NY, USA: ACM, 2006, pp. 73–82.
- [12] D. C. Luckham, "Rapid: A language and toolset for causal event modeling of distributed system architectures," in *WWCA*, 1998.
- [13] 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.
- [14] J.-P. Talpin, C. Brunette, T. Gautier, and A. Gamatié, "Polychronous mode automata," in *EMSOFT*, 2006, pp. 83–92.
- [15] P. A. Abdulla and J. Deneux, "Designing safe, reliable systems using scade," in *In Proc. ISoLA 2004*, 2004.
- [16] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Reading, Massachusetts: Addison-Wesley, 1988.
- [17] 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.
- [18] E. W. Dijkstra, "Guarded commands, nondeterminacy and formal derivation of programs," *Commun. ACM*, vol. 18, no. 8, 1975.
- [19] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," *CWI Quarterly*, vol. 2, pp. 219–246, 1989.
- [20] E. Czeck, R. Nanavati, and J. Stoy, "Reliable Design with Multiple Clock Domains," in *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, 2006.
- [21] *Bluespec SystemVerilog Version 3.8 Reference Guide*, Bluespec, Inc., Waltham, MA, November 2004.
- [22] N. Dave, M. Katelman, M. King, J. Meseguer, and Arvind, "Verification of Microarchitectural Refinements in Rule-based systems," in *MEMOCODE*, Cambridge, UK, 2011.
- [23] 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.
- [24] W. Snyder and P. Wasson, and D. Galbi, "Verilator," <http://www.veripool.com/verilator.html>, 2007.
- [25] "Carbon Design Systems Inc," <http://carbondesignsystems.com>.
- [26] P. Chou, R. Ortega, and G. Borriello, "Synthesis of the hardware/software interface in microcontroller-based systems," in *Proceedings of the International Conference on Computer Aided Design*, 1992, pp. 488–495.
- [27] P. H. Chou, R. B. Ortega, and G. Borriello, "The chinook hardware/software co-synthesis system," in *In International Symposium on System Synthesis*, 1995, pp. 22–27.
- [28] S. Y. Liao, S. W. K. Tjiang, and R. K. Gupta, "An efficient implementation of reactivity for modeling hardware in the scenic design environment," in *DAC*, 1997, pp. 70–75.
- [29] *Catapult-C Manual and C/C++ style guide*, Mentor Graphics, 2004.
- [30] Synfora, "PICO Platform," <http://www.synfora.com/>.
- [31] 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.
- [32] 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, 2008.
- [33] 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.
- [34] 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.
- [35] 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.
- [36] 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>
- [37] N. Dave, Arvind, and M. Pellauer, "Scheduling as Rule Composition," in *Proceedings of Formal Methods and Models for Codesign (MEMOCODE)*, Nice, France, 2007.
- [38] 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)*.
- [39] M. Vijayaraghavan and Arvind, "Bounded dataflow networks and latency-insensitive circuits," in *MEMOCODE*, 2009, pp. 171–180.
- [40] K. Fleming, M. Adler, M. Pellauer, A. Parashar, Arvind, and J. Emer, "Leveraging latency-insensitivity to ease multiple fpga design," in *FPGA*, February 2011.
- [41] T. Esposito, M. Lis, R. Nanavati, J. Stoy, and J. Schwartz, "System and method for scheduling TRS rules," United States Patent US 133051-0001, February 2005.
- [42] J. C. Hoe, "Operation-Centric Hardware Description and Synthesis," Ph.D. dissertation, MIT, Cambridge, MA, 2000.
- [43] M. Karczmarek and Arvind, "Synthesis from multi-cycle atomic actions as a solution to the timing closure problem," in *ICCAD*, 2008.