Hardware-Software Codesign in Bluespec

Arvind
Computer Science & Artificial Intelligence Lab
Massachusetts Institute of Technology

Work in progress: Nirav Dave and Myron King

Ogg Vorbis Pipeline

- Ogg Vorbis is a audio compression format roughly comparable to other compression formats: e.g. MP3, AAC, MWA.
- Input is a stream of compressed bits
- Parsed into frame residues and floor "predictions"
- The summed frequency results are converted to time-valued sequencies
- Final frames are windows to smooth out irregularities
- IMDCT takes the most computation

Ogg Vorbis Pipeline:

- Stream Parser
  - Bits
  - Residue Decoder
    - Floor Decoder
    - IMDCT
      - Windowing
        - PCM Output
Array imdct(int N, Array vx){
    // preprocessing loop
    for(i = 0; i < N; i++){
        vin[i] = convertLo(i,N,vx[i]);
        vin[i+N] = convertHi(i,N,vx[i]);
    }
    // do the IFFT
    vifft = ifft(2*N, vin);
    // postprocessing loop
    for(i = 0; i < N; i++){
        int idx = bitReverse(i);
        vout[idx] = convertResult(i,N,vifft[i]);
    }
    return vout;
}

Suppose we want to use hardware to accelerate FFT/IFFT computation.

January 21, 2011
http://csg.csail.mit.edu/SNU

Implement or find a hardware IFFT.

How will the HW/SW communication work?

How do we explore design alternatives?
HW Accelerator in a system

- Communication via bus
  - DMA transfer?
- Accelerators are all multiplexed on bus
  - Possibly introduces conflicts
  - Fair sharing of bus bandwidth

---

The HW Interface

- SW calls turn into a set of memory-mapped calls through Bus
- Three communication tasks
  - Set size of IFFT
  - Enter data stream
  - Take output out
Data Compatibility Issue

IFFT takes Complex fixed point numbers. How do we represent such numbers in C and in RTL?

```
#include <iostream>

template <typename T>
struct Complex{
    T rel;
    T img;
};
```

```c
typedef struct {
    bit [31:0] fract;
    bit [31:0] integer;
} FixedPt;
```

```
template <typename T>
struct FixedPt{
    T fract;
    T integer;
};
```

```c++
typedef struct {
    FixedPt rel;
    FixedPt img;
} Complex_FixedPt;
```


Data Compatibility

- Keeping HW and SW representation is tedious and error prone
  - Issues of endianness (bit and byte)
  - Layout changes based on C compiler
    - (gcc vs. icc vs. msvc++)
- Some SW representation do not have a natural HW analog
  - What is a pointer? Do we disallow passing trees and lists directly?
- Ideally translation should be automatically generated

Let us assume that data compatibility issue have been solved and focus on control issues
First Attempt at Acceleration

```c
Array imdct(int N, Array<Complex<FixedPt<int,int>> vx){
    // preprocessing loop
    for(i = 0; i < N; i++){
        vin[i] = convertLo(i,N,vx[i]);
        vin[i+N] = convertHi(i,N,vx[i]);
    }
    pcie_ifc.setSize(2*N);
    for(i = 0; i < 2*N; i++)
        pcie_ifc.put(vin[i]);
    for(i = 0; i < 2*N; i++)
        vifft[i] = pcie_ifc.get();
    // postprocessing loop
    for(i = 0; i < N; i++){
        int idx = bitReverse(i);
        vout[idx] = convertResult(i,N,vifft[i]);
    }
    return vout;
}
```

Exposing more details

```c
//mem-mapped hw register
volatile int* hw_flag = ...
//mem-mapped hw frame buffer
volatile int* fbuffer = ...

Array imdct(int N, Array<Complex<FixedPt<int,int>> vx){
    assert(*hw_flag== IDLE);
    for(cnt = 0; cnt<n; cnt++)
        *(fbuffer +cnt)= frame[cnt];
    *hw_flag = GO;
    while(*hw_flag != IDLE) {};
    for(cnt = 0; cnt<n*2; cnt++)
        frame[cnt++]=*(fbuffer+cnt);
```

Issues

- Are the internal hardware conditions correctly exposed correctly by the hw_flag control register?
- Blocking SW is problematic:
  - Prevents the processor from doing anything while the accelerator is in use
  - Hard to pipeline the accelerator
  - Does not handle variation in timing well

Driving a Pipelined HW

```c
int pid = fork();
if(pid){
    // producer process
    while(...){
        ...  
        for(i = 0; i < 2*N; i++)
            pcie.put(vin[i]);
    }
} else {
    // consumer process
    while(...){
        for(i = 0; i < 2*N; i++)
            v[i] = pcie.get();
        ...
    }
}
```

- Multiple processes exploit pipeline parallelism in the IFFT accelerator.
- How does the BSV exert back pressure on the producer thread?
- How does the consumer thread exert back pressure on the BSV module?
- What if our frames are really large, could the HW begin working before the entire frame is transmitted?
Data Parallelism 1

... SyncQueue<Complex<...>> workQ();
int pid = fork();
// both threads do same work
while(...) {
    Complex<FixedPt>* vin = workQ.pop();
    ...
    for(i = 0; i < 2*N; i++)
        pcie.put(vin[i]);
    for(i = 0; i < 2*N; i++)
        v[i] = pcie.get();
    ...
...  
How do we isolate each thread’s use of the HW accelerator?

Do two synchronization points (workQ and the HW accelerator) cause our design to deadlock?


Data Parallelism 2

PCIE get_hw(int pid){
    if(pid==0)
        return pcieA;
    else
        return pcieB;
}
...
SyncQueue<Complex<...>> workQ();
int pid = fork();
// both threads do same work
while(...) {
    Complex<FixedPt>* vin = workQ.pop();
    ...
    for(i = 0; i < 2*N; i++)
        get_hw(pid).put(vin[i]);
    for(i = 0; i < 2*N; i++)
        v[i] = get_hw(pid).get();
    ...
...  
By giving each thread its own HW accelerator, we have further increased data parallelism

If the HW is not the bottleneck this could be a waste of resources.

Do we multiplex the use of the physical BUS between the two threads?

Multithreading without threads or processes

```c
int icnt, ocnt = 0;
Complex iframe[sz];
Complex oframe[sz];
...
// IMDCT loop
while(...){
    // producer “thread”
    for(i = 0; i<2,icnt<n; i++)
        if(pcie.can_put())
            pcie.put(iframe[icnt++]);
    // consumer “thread”
    for(i = 0; i<2,ocnt<n*2; i++)
        if(pcie.can_get())
            oframe[ocnt++] = pcie.get();
    ...
}
```

- Embedded execution environments often have little or no OS support, so multithreading must be emulated in user code.
- Getting the arbitration right is a complex task.
- All existing issues are compounded with the complexity of the duplicated states for each “thread”.

The message

- Writing SW which can safely exploit HW parallelism is difficult...
- Particularly difficult if shared resources (e.g. bus) are involved...
A new approach

- A single language to express the algorithm and indicate a HW/SW partitioning.
- A compiler and run-time to automatically take care of all the ugly bits.
- This language must generate both efficient hardware and low-level software to be of practical use.

BCL: Bluespec Codesign Language [Nirav Dave, Myron King, Arvind]

- BCL is like Bluespec SystemVerilog (BSV) but with extensions for efficient SW specification
  - expressing parallelism comes naturally
- BSV to HW is well understood; use Bluespec Inc.’s commercially available compiler to translate BCL to Verilog
- BCL supports partitioning, giving clear interface semantics between hardware and software domains, which are enforced by the compiler and runtime
- BCL can be written in different styles targeted either at more efficient HW or SW, while always maintaining clear semantics.
We revisit the previous examples, this time in BCL....

First Attempt (BCL)

```
Sync sync <- mkSyncFIFO();
reg cnt <- mkReg(0);
...
rule preprocess when (…)
...
rule fill when (cnt < n);
  sync.toHW(frame[cnt]);
  cnt <= cnt+1;
rule drain when (n<cnt<n*2);
  rv <- sync.fromHW();
  frame[cnt] <- rv;
  cnt <= (cnt <2*n)?cnt+1:0;
rule outp when (n<cnt<2*n);
  rv <- sync.fromHW();
  let x <- ifft.get();
  sync.toSW(x);
  cnt <= (cnt<2*n)?cnt+1:0;
rule postprocess when (…)
...
```

SW partition

HW partition
Advantages

- No data-type compatibility issues; both HW and SW in BCL
- BUS communication completely encapsulated in BCL library modules
- Guarded interfaces are correctly implemented between HW and SW

Driving Pipelined HW (BCL)

```plaintext
Sync sync <- mkSyncFIFO();  Sync sync <- mkSyncFIFO();
reg cnt <- mkReg(0);       IFFT ifft <- mkIFFTPipelined();
rule preprocess when (...)  ...
    ...
rule fill when (icnt<n);    rule inp when (True);
    sync.toHW(iframe[icnt]); let x <- sync.fromSW();
    icnt <= icnt+1;           ifft.put(x);
rule drain when (ocnt < n*2); rule outp when (True);
    rv <- sync.fromHW();     let x <- ifft.get();
    oframe|ocnt| <= rv;       sync.toSW(x);
    ocnt <= ocnt+1;
rule postprocess when (...)  ...
    ...
```

SW partition

HW partition
Driving Pipelined HW (BCL)

- No threads, just parallel rules which the compiler can exploit
- Back pressure from HW to SW is transmitted per the language semantics
- Likewise, back pressure from SW to HW is correctly implemented.

Data Parallelism 1 (BCL)

```haskell
Sync sync <- mkSyncFIFO();
Sync sync <- mkSyncFIFO();
WorkQue wq <- mkWorkQ();
reg cnt <- mkReg(0);
Reg a_tok <- mkReg(True);
IFFT ifft <- mkIFFTPipelined();
Reg b_tok <- mkReg(False);
...
rule a1 when (!b_tok);
  while(cnt<n)
    sync.toHW(aframe[cnt]);
    a_tok <= true;
rule a2 when (a_tok)
  while(cnt<2*n)
    ifft.put(x);
rule inp when;
  let x <- sync.fromSW();
  ifft.put(x);
rule outp when;
  let x <- ifft.get();
  sync.toSW(x);
  aframe[cnt] <= rv;
  cnt <= cnt+1;
rule b1 when (!a_tok);
  HW partition
rule b2 when (b_tok);
  SW partition
```
Data Parallelism 1 (BCL)

- All resources are explicit, and sharing is straightforward
- Synchronization is between a and b is subsumed by rule scheduling
- This implementation is unfair, but changing this is trivial.

Data Parallelism 2 (BCL)

```
Sync synca <- mkSyncFIFO(0);
Sync syncb <- mkSyncFIFO(1);
...  
rule a1 when (True);
while(acnt<n)
    synca.toHW(aframe[cnt]);  
rule a2 when (True);
while(cnt<n+2*n)
    rv <- synca.fromSW();  
    aframe[cnt-n] <= rv;  
    cnt <= cnt+1
...  
rule b1 when (True)
...  
rule b2 when (True)
...  
```

SW partition  HW partition
Data Parallelism 2 (BCL)

- Pipeline and data parallelism in both hardware and software
- BUS is automatically multiplexed to accommodate multiple virtual channels
- As always, resources are explicit.

Some Final Points:

1. There are ways to write rules which will produce efficient SW
2. If the programmer suspects that a particular rule may end up in one specific domain and not the other, it may influence how he defines the rule
3. If the programmer is unsure, it is easy to write the rule in a "target agnostic manner" (recall that no style can violate the BCL semantics of atomicity and guarded interfaces)
4. If you are writing high-level application SW way up the stack, use C++, don’t use BCL