Mapping High-Level Language Applications to SystemC

Steven P. Smith
Design of Embedded Systems
EE382V
Fall 2009

Overview

• Many projects begin with a working HLL application.
  • Particularly common in multimedia and communications
• Goals
  • Maintain conformance with original application software throughout development
    • The HLL implementation serves as the gold model.
  • Profiling data and architecture considerations drive process
• Support incremental mapping of modules
  • One accelerator at a time…
• SystemC is well suited for use with C/C++ HLL apps.
Standards-Based Applications

- Predominant in some important domains
  - Wireless Communications
  - Networking
  - Audio, Video
- Standards are often developed using a community-developed HLL application.
  - The application embodies the standard.
- Such *reference* applications are seldom well structured or optimized.
- Full conformance to standard is crucial to success.

Steps in Moving from Reference Code to an Embedded Implementation

- Algorithm study and analysis
- Setting target performance requirements
- Understanding reference code structure
- Profiling, identification of bottlenecks
  - What functions must be accelerated to meet performance targets?
- Developing a Hardware Abstraction Layer (HAL)
- Modifying code to make it suitable for hardware
- Mapping the application onto the hardware
- Readying the application for production
From Reference Code to Embedded Implementation

- Conformance maintained at all phases of design
- Reference application used to generate module-level test vectors
- Hardware/Software co-design a natural by-product
- Hardware abstraction layer (HAL) defines the interface between application software and any special-purpose hardware
- Need a development environment supporting hardware models and application software execution
  - SystemC (informal software support)

SystemC

- Class library extension to C++
- Extended to support verification-specific constructs
- C++ can be intimidating to HW designers trained in Verilog or VHDL
- Software developers find it easier to integrate their programs and tools than with other ESL languages.
- Open standard effort through the Open SystemC Initiative (OSCI)
- Synthesis tools emerging in the marketplace (e.g., Forte, Mentor Graphics)
Considerations in Selecting Modules for Realization as Accelerators

- Software profile data can help filter candidate modules.
- HLL function boundaries not always appropriate module boundaries
  - Some code refactoring may be necessary
- Input/output requirements also a factor
  - Transfer overhead can swamp advantage of acceleration
- Look for opportunities for module-level parallelism
  - Identify synchronization requirements
- Global variables must be eliminated from module
  - Map to arguments
- Data transfer alternatives (e.g., DMA, processor-directed)
The Hardware Abstraction Layer (HAL)

• Provides an efficient interface to hardware while maintaining application code structure.
  • From application, HW accelerator looks like a function call
  • From HW accelerator, application looks like HW/buffer
• Define functions for each HW/SW interaction, isolating hardware detail from application software.
• Provide synchronization primitives required for flow control, management of parallel activity.
• Enables mixture of hardware and software models
  • Selective use of hardware modules supports debug in emulation environment
  • Expanded verification challenge

Formalizing the HAL Interfaces

• A formal, standardized representation would be helpful.
  • Support tool-based interface generation and error-checking
  • Ease IP re-use
  • Capture constraints
  • Bridge the gap between software and hardware constraints
• The Architecture and Analysis and Design Language (AADL)
  • Originally the “Avionics Architecture Description Language”
  • Safety-critical and mission-critical applications were initial focus
  • Separates types (interfaces) and implementations
  • http://www.sei.cmu.edu/pub/documents/06.reports/pdf/06tn011.pdf
HAL Application Interface Layer

• Application Layer
  • Maps HLL function call to lower layer of HAL

```c
void appFunction1(int * data, int dataSize) {
    #if HAL_ENABLED
        int i;
        HAL_checkReadyFunction1(TRUE);
        for (i=0; i < dataSize; i++) {
            HAL_enqueueToFunction1(data[i]);
        }
        HAL_startFunction1();
    #else
        // ... Existing HLL application function code ...
    #endif
}
```

HAL Hardware Interface Layer

• Hardware Layer
  • Interacts with HW for synchronization, data transfers, and status queries

```c
int HAL_waitReadyFunction1(int waitTillReady) {
    int result;
    do {
        result = HAL_ADR_Function1Status();
    } while (!result && waitTillReady);
    return result;
}
```
HAL Layers

- Hardware Layer with Instrumentation
- Can be used to capture stimuli

```c
int HAL_enqueueToFunction1(int data)
{
#if HAL_GEN_VERILOG
    fprintf(pfVStim,"halFunction1InputData = %d ;\n", data);
#else
    halFunction1InputQueue = data;
#endif
}
```

Example - MPEG-2 Video Decode

Audio/Video Demultiplex → Video Bitstream Decode → Inverse Discrete Cosine Transform → Inverse Motion Prediction → Frame Store → Audio Decode

- Error Coefficients
- Predicted Pixels
- Transform Coefficients
- Motion Vectors
Inverse Discrete Cosine Transform

- Fundamental process in most image compression algorithms
  - JPEG, MJPEG, MPEG
- Image data tends to show correlation in frequency domain.
- Forward discrete cosine transform (DCT) used during encoding
  - Intra (I) frames: coefficients are frequency-domain pixel values
  - Prediction (P) frames: coefficients are prediction errors
- Inverse discrete cosine transform (iDCT) used during decoding to recover image
- Color components processed separately

Inverse Discrete Cosine Transform

- Processing iDCTs becomes time critical for video decoding.

Consider the case of 1080i HD video

- Pixels per frame: 1088\(^1\) \times 1920 = 2,088,960
- Frame rate: 30 frames per second\(^2\)
- Required two-dimensional 8x8 iDCTs per second: 
  \[
  \frac{(2088960 \times 30)}{(8 \times 8)} = 244,800 \text{ iDCTs/second for luminance data alone}
  \]
- Time budget for each luminance iDCT: 4.08 uSecs (2.7 uSecs with chroma data added)

---

\(^1\) Height padded to 1088 to be multiple of 16.
\(^2\) Actually 60 interlaced fields per second
MPEG-2 iDCT Reference Software

#define BLSIZE 8

void dct_two_d(short int **in, short int **coeff)
{
    register int    j1, i, j, offset;
    float          b[BLSIZE], c[BLSIZE], d[BLSIZE][BLSIZE];

    for (i = offset = 0; i < 8; i++)
    {
        for (j = 0; j < 8; j++)
            b[j] = (float) (in[i][j]);

        /* Horizontal transform */
        for (j = 0; j < 4; j++)
        {
            j1 = 7 - j;
            c[j] = b[j] + b[j1];
            c[j1] = b[j] - b[j1];
        }
        b[0] = c[0] + c[3];
        b[3] = c[0] - c[3];
        b[4] = c[4];
        b[5] = (c[6] - c[5]) * f0;
        b[6] = (c[6] + c[5]) * f0;
        b[7] = c[7];
        d[i][0] = (b[0] + b[1]) * f4;
        d[i][4] = (b[0] - b[1]) * f4;
    }

    /* Vertical transform */
    for (i = 0; i < 8; i++)
    {
        for (j = 0; j < 4; j++)
        {
            j1 = 7 - j;
            c[j] = d[j][i] + d[j1][i];
            c[j1] = d[j][i] - d[j1][i];
        }
    }
}
MPEG-2 iDCT Reference Software (3)

\[
\begin{align*}
b[0] &= c[0] + c[3]; \\
b[3] &= c[0] - c[3]; \\
b[4] &= c[4]; \\
b[5] &= (c[6] - c[5]) \times f0; \\
b[6] &= (c[6] + c[5]) \times f0; \\
b[7] &= c[7]; \\
d[0][i] &= (b[0] + b[1]) \times f4; \\
d[4][i] &= (b[0] - b[1]) \times f4; \\
d[2][i] &= b[2] \times f6 + b[3] \times f2; \\
d[6][i] &= b[3] \times f6 - b[2] \times f2; \\
d[1][i] &= c[4] \times f7 + c[7] \times f1; \\
d[5][i] &= c[5] \times f3 + c[6] \times f5; \\
d[7][i] &= c[7] \times f7 - c[4] \times f1; \\
\end{align*}
\]

MPEG-2 iDCT Reference Software (4)

// Do rounding instead of just truncating.
// Decided in 38.MPEG-meeting in Sevilla
// Note: rounding is for accurate reference
// If for speed encoding, you may go without this.
// There is no discernible effect in image quality.
for (i=0;i<8;i++)
    for (j=0;j<8;j++)
        if(d[i][j] >=0) {
            coeff[i][j] = (short int)(d[i][j] +
            0.499999999999);  
        }
        else {
            coeff[i][j] = (short int)(d[i][j] -
            0.499999999999);  
        }

    // clipping range
    if(coeff[i][j] < -2048) coeff[i][j] = -2048;
    if(coeff[i][j] > 2047) coeff[i][j] = 2047;
}
Observations on the Reference Code

• Floating point is expensive in hardware
• Highly sequential computation as structured
  • Two-dimensional control loop doesn’t take advantage of observation that the vertical and horizontal operation sequences are identical
  • Opportunity to re-use hardware and/or software for both directions
• Final rounding operation and clipping pass over data is time-consuming
• Rounding and clipping pass highly sequential in general-purpose hardware

Conversion of Floating Point Arithmetic Operations to Integer Arithmetic

• Determine dynamic range of floating point calculations, including all intermediate values
  • Analytical approach superior when possible
  • Random and directed test vectors with range accumulation on each variable where direct analysis is not practical
• If range is too large
  • Consider pre-scaling data to narrower range.
    • Must determine if error introduced is acceptable
  • Re-scale the results at end of computation
  • Restructure/reorder arithmetic operations to reduce dynamic range for intermediate calculations.
Modifying the iDCT Code for Hardware

- Integer-only computation needed
- Mapping the reference code directly into an integer representation
  - Dynamic range of data within 32-bit integer capacity
  - Only barely
- Can the algorithm be modified to reduce the dynamic range required?
- Can 16-bit intermediate values suffice?
- What loss of accuracy will occur?

MPEG-2 Integer iDCT Software

```c
#define W1 2841 /* 2048*sqrt(2)*cos(1*pi/16) */
#define W2 2676 /* 2048*sqrt(2)*cos(2*pi/16) */
#define W3 2408 /* 2048*sqrt(2)*cos(3*pi/16) */
#define W5 1609 /* 2048*sqrt(2)*cos(5*pi/16) */
#define W6 1108 /* 2048*sqrt(2)*cos(6*pi/16) */
#define W7 565  /* 2048*sqrt(2)*cos(7*pi/16) */

/* row (horizontal) IDCT
 *           7                       pi         1
 * dst[k] = sum c[l] * src[l] * cos( -- * ( k + - ) * l )
 *          l=0                      8          2
 * where: c[0] = 128
 *        c[1..7] = 128*sqrt(2)
 */
static void idctrow(short *blk)
{
    int x0, x1, x2, x3, x4, x5, x6, x7, x8;

    if (!((x1 = blk[4]<<11) | (x2 = blk[6]) | (x3 = blk[2]) |
         (x4 = blk[1]) | (x5 = blk[7]) | (x6 = blk[5]) | (x7 = blk[3])))
    {
        return;
    }
```
MPEG-2 Integer iDCT Software (2)

x0 = (blk[0]<<<11) + 128; /* for proper rounding in the fourth stage */

/* first stage */
x8 = W7*(x4+x5);  
x4 = x8 + (W1-W7)*x4;  
x5 = x8 - (W1+W7)*x5;  
x8 = W3*(x6+x7);  
x6 = x8 - (W3-W5)*x6;  
x7 = x8 - (W3+W5)*x7;

/* second stage */
x8 = x0 + x1;  
x0 -= x1;  
x1 = W6*(x3+x2);  
x2 = x1 - (W2+W6)*x2;  
x3 = x1 + (W2-W6)*x3;  
x1 = x4 + x6;  
x4 -= x6;  
x6 = x5 + x7;  
x5 -= x7;

MPEG-2 Integer iDCT Software (3)

/* third stage */
x7 = x8 + x3;  
x8 -= x3;  
x3 = x0 + x2;  
x0 -= x2;  
x2 = (181*(x4+x5)+128)>>8;  
x4 = (181*(x4-x5)+128)>>8;

/* fourth stage */
blk[0] = (x7+x1)>>8;  
blk[1] = (x3+x2)>>8;  
blk[2] = (x0+x4)>>8;  
blk[3] = (x8+x6)>>8;  
blk[4] = (x8-x6)>>8;  
blk[5] = (x0-x4)>>8;  
blk[6] = (x3-x2)>>8;  
blk[7] = (x7-x1)>>8;  
}
MPEG-2 Integer iDCT Software (4)

/* column (vertical) IDCT */
7
* dst[8*k] = sum c[l] * src[8*l] * cos( -- * ( k + - ) * 1 )
* l=0
* 8
* 2
* where: c[0] = 1/1024
*        c[1..7] = (1/1024)*sqrt(2)
*/
static void idctcol(short *blk)
{
    int x0, x1, x2, x3, x4, x5, x6, x7, x8;
    /* shortcut */
    if (!((x1 = (blk[8*4]<<8)) | (x2 = blk[8*6]) | (x3 = blk[8*2]) |
        (x4 = blk[8*1]) | (x5 = blk[8*7]) | (x6 = blk[8*5]) |
        (x7 = blk[8*3])))
    {
        blk[8*0]=blk[8*1]=blk[8*2]=blk[8*3]=
            gVideoData->iclp[(blk[8*0]+32)>>6];
        return;
    }
    x0 = (blk[8*0]<<8) + 8192;
    /* first stage */
    x8 = W7*(x4+x5) + 4;
    x4 = (x8+(W1-W7)*x4)>>3;
    x5 = (x8-(W1+W7)*x5)>>3;
    x8 = W3*(x6+x7) + 4;
    x6 = (x8-(W3-W5)*x6)>>3;
    x7 = (x8-(W3+W5)*x7)>>3;
    /* second stage */
    x8 = x0 + x1;
    x0 = x1;
    x1 = W6*(x3+x2) + 4;
    x2 = (x1-(W2+W6)*x2)>>3;
    x3 = (x1+(W2-W6)*x3)>>3;
    x1 = x4 + x6;
    x4 = x6;
    x6 = x5 + x7;
    x5 = x7;

MPEG-2 Integer iDCT Software (5)

/* first stage */

x8 = W7*(x4+x5) + 4;

x4 = (x8+(W1-W7)*x4)>>3;

x5 = (x8-(W1+W7)*x5)>>3;

x8 = W3*(x6+x7) + 4;

x6 = (x8-(W3-W5)*x6)>>3;

x7 = (x8-(W3+W5)*x7)>>3;

/* second stage */

x8 = x0 + x1;

x0 = x1;

x1 = W6*(x3+x2) + 4;

x2 = (x1-(W2+W6)*x2)>>3;

x3 = (x1+(W2-W6)*x3)>>3;

x1 = x4 + x6;

x4 = x6;

x6 = x5 + x7;

x5 = x7;
MPEG-2 Integer iDCT Software (6)

/* third stage */
x7 = x8 + x3;
x8 -= x3;
x3 = x0 + x2;
x0 -= x2;
x2 = (181*(x4+x5)+128)>>8;
x4 = (181*(x4-x5)+128)>>8;

/* fourth stage */
blk[8*0] = gVideoData->iclp[(x7+x1)>>14];
blk[8*1] = gVideoData->iclp[(x3+x2)>>14];
blk[8*2] = gVideoData->iclp[(x0+x4)>>14];
blk[8*3] = gVideoData->iclp[(x8+x6)>>14];
blk[8*4] = gVideoData->iclp[(x8-x6)>>14];
blk[8*5] = gVideoData->iclp[(x0-x4)>>14];
blk[8*6] = gVideoData->iclp[(x3-x2)>>14];
blk[8*7] = gVideoData->iclp[(x7-x1)>>14];

MPEG-2 Integer iDCT Software (7)

/* two dimensional inverse discrete cosine transform */
void Fast_IDCT(short **inblock, short** outblock)
{
    short *block =  *inblock;

    int i;

    for (i=0; i<8; i++)
        idctrow(block+8*i);

    for (i=0; i<8; i++)
        idctcol(block+i);
}
Comments on the Integer iDCT Variant

- Only slight loss of accuracy (>40 dB pSNR for iDCT)
  - Anything > ~30 dB pSNR is usually deemed okay for video
- Use of 16-bit integers allows efficient mapping to MMX instructions for faster execution on a general-purpose machine.
- Shifts are efficiently implemented in hardware.
- Opportunity for early cut-off in software
- Note similarity of horizontal and vertical processing

Options for Accelerating the iDCT

- Simplest: use existing MMX instructions on PC
- Highest performance: map into 256-input, 256-output maximally parallel logic
  - Fast, but very large
  - No flexibility
Options for Accelerating the iDCT

- High performance: translate the simple C to Verilog, synthesize one-dimensional vector iDCT, replicate the module 8 times for each dimension
  - Simpler than maximally parallel concept
  - Still no flexibility

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

Row Col

---

Options for Accelerating the iDCT

- High performance but smaller: reuse one-dimensional slices for row and column
  - “Transpose memory” serves as buffer
  - Still no flexibility

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

TM
Options for Accelerating the iDCT

- High performance with increased flexibility: microcode engine with SIMD
- “Transpose memory” serves as buffer
- Works on 8x8 (or smaller) blocks for iDCT, DCT, forward and inverse integer transform (H.264)

Integer iDCT for Microcode Engine

```c
static void idct_x2(int i0, int i1, int i2, int i3, /* input values */
    int i4, int i5, int i6, int i7,
    int j0, int j1, int j2, int j3,
    int j4, int j5, int j6, int j7,
    char isRow, /* non-zero if row */
    int * o0, int * o1, int * o2, int * o3, /* outputs */
    int * o4, int * o5, int * o6, int * o7,
    int * p0, int * p1, int * p2, int * p3,
    int * p4, int * p5, int * p6, int * p7)
{
    int r0, r1, r2, r3, r4, r5, r6, r7 ; /* 1 */
    r0  = i1 * C7 ; /* 2 */
    r1  = i7 * C1 ; /* 3 */
    r2  = i5 * C3 ; /* 4 */
    r3  = i3 * C5 ;
    r0  = (r0 - r1) >> (isRow ? 0 : 8) ; /* 5 */
    r1  = i7 * C7 ;
```
Integer iDCT for Microcode Engine (2)

/* 6 */
  r3 = i1 * C1 ;
  r2 = (r2 - r3) >> (isRow ? 0 : 8) ; /* f */
/ * 7 */
  r7 = i3 * C3 ;
  r2 = (r0 + r2) ; /* b[4] */
  r0 = r0 + r2 ; /* b[5] */
/ * 9 */
  r7 = i6 * C6 ;
  r1 = (r1 + r3) >> 3 ; /* b[6] */
/ * 10 */
  r5 = i2 * C2 ;
  r3 = (r7 + r3) >> (isRow ? 0 : 8) ; /* g */
/ * 11 */
  r7 = i6 * C6 ;
  r1 = (r1 - r3) >> 3 ; /* b[7] */
/ * 12 */
  r4 = i6 * C6 ;
  r2 = (r0 - r2) >> 3 ; /* b1[5] */
/ * 13 */
  r6 = (r6 - r7) >> (isRow ? 0 : 8) ; /* b1[2] */
  r7 = i0 + i4 ;
/ * 14 */
  r7 = (r7 * C4) >> (isRow ? 0 : 8) ; /* b1[0] */
  r2 = r1 + r2 ;
  r5 = i0 - i4 ;
/ * 15 */
  r5 = (r5 * C4) >> (isRow ? 0 : 8) ; /* b1[1] */
  r7 = r7 + r4 ; /* b[0] */
  r4 = r7 - r4 ; /* b[3] */
/ * 16 */
  r1 = (r1 * C0) >> 8 ; /* b[5] */
  r5 = r5 + r6 ; /* b[1] */
  r6 = r5 - r6 ; /* b[2] */
/ * 17 */
  r2 = (r2 * C0) >> 8 ; /* b[6] */

Integer iDCT for Microcode Engine (3)

/* 13 */
  r6 = (r6 - r7) >> (isRow ? 0 : 8) ; /* b[2] */
  r7 = i0 + i4 ;
/ * 14 */
  r7 = (r7 * C4) >> (isRow ? 0 : 8) ; /* b1[0] */
  r2 = r1 + r2 ;
  r5 = i0 - i4 ;
/ * 15 */
  r5 = (r5 * C4) >> (isRow ? 0 : 8) ; /* b1[1] */
  r7 = r7 + r4 ; /* b[0] */
  r4 = r7 - r4 ; /* b[3] */
/ * 16 */
  r1 = (r1 * C0) >> 8 ; /* b[5] */
  r5 = r5 + r6 ; /* b[1] */
  r6 = r5 - r6 ; /* b[2] */
/ * 17 */
  r2 = (r2 * C0) >> 8 ; /* b[6] */
Mapping the Hardware Functions to SystemC Models

- HAL separates application software from SystemC “collars” (application-layer HAL) on functions to be implemented in hardware
- Initially, SystemC model is nothing more than functional HLL code with ports.
- Follow-on with successive refinements to model reflecting hardware structural decomposition.
- Useful for simulation-based debugging of microcode engines
- Alternatively, refine model to synthesizable form
- All models verified against reference software
SystemC HAL Interface for Integer iDCT

```c
#include "systemc.h"
SC_MODULE(idct) // declare iDCT sc_module
{
sc_in_clk ck ;
sc_in<sc_bv<32>> indata ; // input signal ports
sc_in<sc_bv<8>> addr ;
sc_in<sc_bit>  wr, sel, go ;
sc_out<sc_bv<32>> outdata ;

unsigned int tm[BLSIZE][BLSIZE] ; // transpose memory

// Interface to the bus
if (sel)
{
    if (wr)
    {
        tm[addr>>4][addr&0xF] = indata ;
    }
    else if (!go)
    {
        outdata = tm[addr>>4][addr&0xF] ;
    }
}
else
{
    int i ;
    for (i=0; i < 8 ; i++)
    {
        // Call behavioral function for rows
        idct_x2(tm[i][0], tm[i][1], tm[i][2], tm[i][3],
                tm[i][4], tm[i][5], tm[i][6], tm[i][7],
                &tm[0][i], &tm[1][i], &tm[2][i], &tm[3][i],
                &tm[4][i], &tm[5][i], &tm[6][i], &tm[7][i]) ;
    }
    for (i=0; i < 8 ; i++)
    {
        // Call behavioral function for columns
        idct_x2(tm[0][i], tm[1][i], tm[2][i], tm[3][i],
                tm[4][i], tm[5][i], tm[6][i], tm[7][i],
                &tm[i][0], &tm[i][1], &tm[i][2], &tm[i][3],
                &tm[i][4], &tm[i][5], &tm[i][6], &tm[i][7]) ;
    }
}
}
```
Things to Consider

- Application profiling with SystemC models presents a challenge.
  - “Back out” profile data and overhead for hardware models
  - Determine cycle counts of the hardware models
  - Build spreadsheet to compute adjusted profile results with clock cycle as parameter
- An instrumented HAL can be useful in assessing system performance.
  - Track bus activity, memory reference patterns, bandwidth requirements.

Conclusions

- For target applications based on reference software, or for any project that begins with "gold" application software, SystemC can provide a smooth path from software to embedded system.
- Select candidate models based on profile data or by known performance requirements (e.g., frame rates for video decode)
- Convert floating point operations to integer to reduce HW complexity and processing time
- Consider interface issues
  - Polling or interrupt for control events
  - DMA or processor-directed transfers
Conclusions (continued)

• Map functions to hardware by first “collaring” them with port interfaces around the pure behavior.

• Increase detail and accuracy of SystemC model as design progresses.

• Use reference application as test bench throughout.

• Use HAL to hide hardware detail from application and to instrument the model.

• Maintain a working model that matches the gold model throughout the entire design.