The Challenges of Hardware Synthesis from C-like Languages

Stephen A. Edwards
Department of Computer Science, Columbia University
www.cs.columbia.edu/~sedwards
sedwards@cs.columbia.edu
Why C?

“A single language would facilitate the step-by-step refinement of a system design down to its components”
[SystemC: Liao et al. 1997]

“All examples contributed by industry were written in the C programming language”
[SpecC: Gajski et al., 2000]

“If you are familiar with conventional C you will recognize nearly all the other features.”
[Handel-C: Celoxica, 2003]
Why Hardware?

Efficiency: Power, speed, or cost.

This talk assumes we have decided to produce hardware.
Genesis: BCPL begat B begat C

BCPL: Martin Richards, Cambridge, 1967
Typeless: everything a machine word
Memory: undifferentiated array of words
Then, processors mostly word-addressed

```
LET try(ld, row, rd) BE TEST row=all
THEN count := count + 1
ELSE $(
    LET poss = all & NOT (ld | row | rd)
    UNTIL poss=0 DO $(
        LET p = poss & ~poss
        poss := poss - p
        try(ld+p << 1, row+p, rd+p >> 1)
    )
) $
```

Part of the N-queens problems implemented in BCPL
C History

Developed 1969–1973 along with Unix
Due mostly to Dennis Ritchie
Designed for systems programming:
operating systems, utility programs, compilers

PDP-11/20 (c. 1970) 24K of core (12K for kernel)
Euclid’s Algorithm on the PDP-11

```c
int gcd(int m, int n)
{
    int r;
    while ((r = m%n) != 0) {
        m = n;
        n = r;
    }
    return n;
}
```

```
.globl _gcd
.text
_gcd:
    jsr r5, rsave
L2:  mov 4(r5), r1
    sxt r0
    div 6(r5), r0
    mov r1, -10(r5)
    jeq L3
    mov 6(r5), 4(r5)
    jbr L2
L3:  mov 6(r5), r0
    jbr L1
L1:  jmp rretrn
```
The Five Big Challenges

- Concurrency
- Timing
- Types
- Communication
- Meta-Data
Traditional C Concurrency: Pthreads

```c
pthread_mutex_t mymutex; /* Mutual Exclusion Variable */
int myglobal = 0; /* Global variable */
pthread_t thread[3]; /* Information about threads */

void *myThread(void *arg) {
    pthread_mutex_lock(&mymutex); /* Get the lock */
    ++myglobal; /* Update shared variable */
    pthread_mutex_unlock(&mymutex); /* Release the lock */
    pthread_exit((void*) 0);
}

void count_to_three() {
    int i, status;
    pthread_attr_t attr;
    pthread_mutex_init(&mymutex, NULL);
    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);
    for (i = 0 ; i < 3 ; i++)
        pthread_create(&thread[i], &attr, myThread, (void *)i);
    for (i = 0 ; i < 3 ; i++)
        pthread_join(thread[i], (void **) &status);
    }
```
Approach 1: Add Parallel Constructs

HardwareC, SystemC, Oapi, Handel-C, SpecC, Bach C

/* Handel-C code for a four-place queue */

void main(chan (in) c4 : 8,
           chan (out) c0 : 8)
{
    int d0, d1, d2, d3;
    chan c1, c2, c3;

    void e0() { while (1) { c1 ? d0; c0 ! d0; } }
    void e1() { while (1) { c2 ? d1; c1 ! d1; } }
    void e2() { while (1) { c3 ? d2; c2 ! d2; } }
    void e3() { while (1) { c4 ? d3; c3 ! d3; } }

    par {
        e0(); e1(); e2(); e3();
    }
}
2: Let Compiler Find Concurrency

Cones, Transmogrifier C, C2Verilog, CASH

/* CONES code counts ones */
INPUTS: IN[5];
OUTPUTS: OUT[3];
rd53() {
    int count, i;
    count = 0;
    for (i = 0 ; i < 5 ; i++ )
        if (IN[i] == 1)
            count = count + 1;
    for (i = 0 ; i < 3 ; i++ ) {
        OUT[i] = count & 0x01;
        count = count >> 1;
    }
}

Compiler unrolls loops

 Fundamental limits on how much concurrency could ever be found [David Wall 91, 94]

This problem: a Holy Grail of Computer Science
Timing in Algorithmic Languages

Algorithm: “A sequence of steps designed to solve a problem.”

Powerful abstraction; inadequate for hardware
Approach 1: Explicit Clocks

Ocapi, SpecC, Cones, SystemC

/* SystemC code for a simple protocol */
while( index < 16 ) {
    data_req.write(true);
    wait_until(data_valid.delayed() == true);
    tmp_real = in_real.read();
    tmp_imag = in_imag.read();
    real[index] = tmp_real;
    imag[index] = tmp_imag;
    index++;
    data_req.write(false);
    wait();
}

Quite a departure
Approach 2: Constraints

HardwareC, C2Verilog

An awkward way to describe behavior

/* Constraints in HardwareC */

constraint maxtime from label1 to label3 = 4 cycles;
constraint delay of label2 = 2 cycles;

label1:
    Y = read(X);
    Y = Y + 1;
label2:
    Y = Y * Q;
label3:
    send(channelA, Y);
Approach 3: Rules Imply Clocks

Handel-C (assignment = clock),
Transmogrifier C (loop iteration = clock),
C2Verilog (complex)

```c
/* Handel-C Transmogrifier C */
for (i = 0 ; i < 8 ; i++ ) {
    /* 9 8 */
    a[i] = c[i];
    /* 8 0 */
    b[i] = d[i] || f[i];
    /* 8 0 */
}

Unwieldy
```
Types

BCPL: everything is a word (word-addressed memory)

C: chars, shorts, ints, longs, floats, doubles (PDP-11’s byte-addressed memory)

Bit-level granularity natural for hardware.
Approach 1: Annotations/External

C2Verilog, Transmogrifier C

/* Selecting bit widths in Transmogrifier C */
#pragma intbits 4
int xval, yval;

#pragma intbits 1
int ready;

Awkward. C2Verilog had a GUI for adding annotations.
Approach 2: Add Hardware Types

HardwareC, Handel-C, Bach C, SpecC

/* Bach C hardware data types */

int#24 a = (101*100)/2;
unsigned#16 b = 1;

while (a)
    a -= b++;

A big change for C programmers
Approach 3: Use C++’s Type System

SystemC, Ocapí

/* Hardware data types in SystemC */
struct fft: sc_module {
    sc_in<sc_int<16> > in_real;
    sc_in<sc_int<16> > in_imag;
    sc_in<bool> data_valid;
    sc_in<bool> data_ack;
    sc_out<sc_int<16> > out_real;
    sc_out<sc_int<16> > out_imag;
    sc_out<bool> data_req;
    sc_out<bool> data_ready;
    sc_in_clk CLK;

    SC_CTOR(fft) {
        SC_CTHREAD(entry, CLK.pos());
    }

    void entry();
};
Communication

Software

Hardware
Communication: Pointers

Assumes a monolithic memory model.

Semeria and De Micheli [ICCAD 2001] used pointer analysis to break memory into separate spaces.

Not implemented in any commercial compiler.
Approach 1: Preserve the C model

CASH, Handel-C, C2Verilog

/* Source C code */
int *p;
struct { int i; short sh[2]; } s;
int b[5];

if (...)  
    p = &s.i;
else
    p = &b[2];
p = p + 1;

out = *p;

P can point into s or into b

/* After Semeria et al. */
int pp;
short sh[4];
int b[5];

if (...)  
    pp = 0 << 16 | 0;
else
    pp = 1 << 16 | 8;
pp = pp + 4;

if ( pp >> 16 == 0 )  
    out = sh[ pp&0xffff >> 1 ] << 16 |  
        sh[ pp&0xffff >> 1 + 1];
else
    out = b[ pp&0xffff >> 2 ];
**Approach 2: Use Other Primitives**

HardwareC (rendezvous)
Handel-C (rendezvous)
Bach C (rendezvous)
SpecC (variety)
SystemC (variety)

```c
/* Handel-C serial-to-parallel */
while (1) {
    bitstream ? bits_0;
    bitstream ? bits_1;
    bitstream ? bits_2;
    bitstream ? bits_3;
    bitstream ? bits_4;
    bitstream ? bits_5;
    bitstream ? bits_6;
    bitstream ? bits_7;
    STDOUT ! bits_0 @ bits_1 @
    bits_2 @ bits_3 @
    bits_4 @ bits_5 @
    bits_6 @ bits_7;
}
```
int g[15];

a = b + c;
d = e + f;

How to implement the “+”?
How many adders?
How should the g array be implemented?
How do you tell the synthesizer what you want?
5.2. Clock Selection and resource allocation

The maximum execution time of a design can be defined as the product of the clock period used in the design and the maximum number of clock periods used in the design and the maximum number of clock cycles. Hence, to optimize the performance of a design, it is important to select the clock period wisely, as well as to minimize the number of clock cycles \[JGC96\]. Moreover, the number of clock cycles required to finish all operations in a design depends on the clock period. Therefore, a bad choice of the clock period could severely affect the performance of the design. In our methodology, the clock selection is done by the user.

Resource allocation is also an important step in RTL synthesis. The number of resources can be determined by automatic tools or by the user \[GDLW92\]. In our RTL design methodology, resource allocation is performed by the user.

5.3. Scheduling and binding algorithm

We describe the simultaneous scheduling and binding algorithm that solves the scheduling and binding problems together. This algorithm is greedy but simple and easy to implement. However, our methodology is independent of scheduling and binding algorithms and can use any other algorithms such as force-directed heuristic as well.

5.3.1. Problem Definition

Given:
1. A behavior represented by a state transition graph, \(STG(S, T)\), where \(S\) is a state in FSMD and \(T\) is a state transition among states.
2. Each state \(S\) contains a hierarchical control/data flow graph, \(CDFG(V, E)\), where \(V\) is a set of vertices representing operations, storages, buses, and hierarchical nodes such as branch and loop, and \(E\) is the dependency between nodes.
3. A component library containing functional units, storages, and buses characterized by type, area, delay, pipeline states, and so on. In addition, storages have the number of read/write ports.
4. Clock period and resource allocation, such as the number of functional units, storage units, buses, and read/write ports of storage units.

Determine:
1. Control step of each node in the behavior

```c
instance counter value1, value2;
instance fastcounter value3;

value1(...); /* first counter */
value1(...); /* first counter */
value2(...); /* second counter */
value3(...); /* third fastcounter */
value2(...); /* second counter */
```
Summary

Concurrency: Explicit or compiler’s job

Timing: Explicit, constraints, or rules

Types: Annotations, additional, C++

Communication: C-like or additional

Meta-Data: GUI or annotations
The next language should have...

- High-level abstractions that address complexity
  Concurrency + communication, timing control, hardware types, and support for refinement
- Constructs that match what designers want
  Datapaths, controllers, memories, busses, hierarchy
- Semantics with an efficient translation into hardware
- Semantics that facilitate very efficient simulation

Will it be like C? At most only superficially.