

Sketching for Multi-core Programming

Apollo Ellis
UC Berkeley

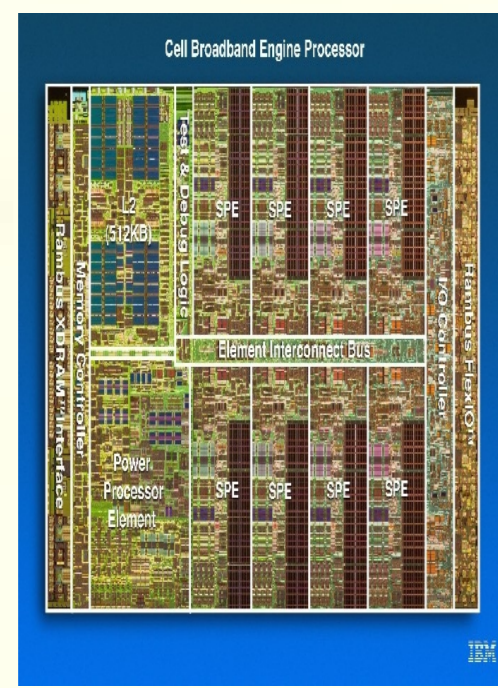
Research Goals

Become associated with the IBM Cell BE Processor in a Playstation 3

Get experience with large multi-core Cell projects ie. ray tracers, game engines etc.

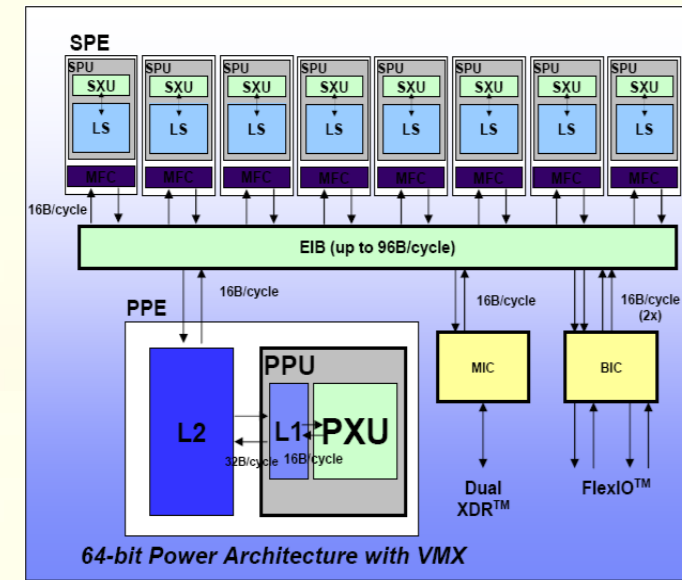
Assist the port of Sketch Synthesis to multi-core sketching

Retire as rich kings in the land of multi-core programming



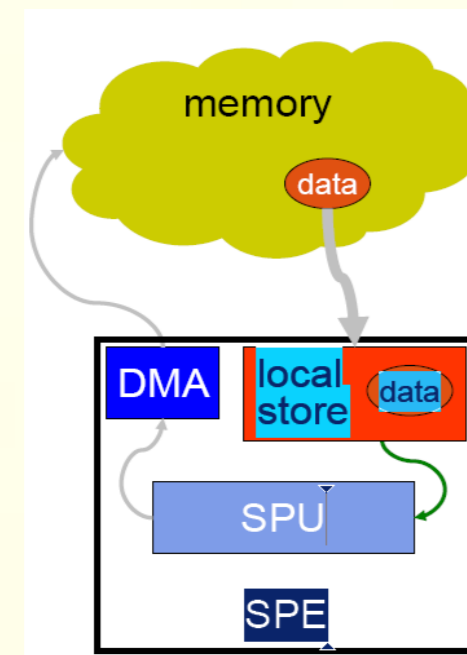
Cell Architecture

- Power PC Unit (PPU) 3.2Ghz
- VMX 32 instruction set
- Control of eight synergistic processing units SPUs
- Main 512KB L2 Cache
- Synergistic Processing Unit (SPU) 3.2Ghz
- Optimized for data streaming
- 128 bit SIMD instruction set
- 256KB local store
- 128x128 bit register file



DMA (Direct Memory Access)

Controlled by the SPU MFC (Memory Flow Controller) unit.
Transfers data directly to and from the PPC L2 Cache and the SPU local store
Transfers data from SPU to SPU
Transfers must be from 1 byte to 16KB by powers of 2 bytes.
No other sized transfers are supported.
DMA bandwidth on the SPU is 128B/cycle.
Request queuing for 8 requests
Bandwidth of the Bus 96B/cycle.



spu_mfcio.h

```
//SPU Initiates DMA transfer from PPU
mfc_get(&addr_of_local_data, addr_of_remote_data, data_size,
MFC_TAG, ...);
//Wait for DMA to finish
mfc_read_tag_status_all();
...Process Data...
//SPU Initiates DMA transfer to PPU
mfc_put(addr_of_local_data, addr_of_remote_data, data_size,
MFC_TAG,...);
// Wait for DMA to finish
mfc_read_tag_status_all();
```

SIMD C/C++ Language intrinsics

```
Vector types are native 128bit quad words:
vector<int> int_vec={1,2,3,4};
vector<char>
char_vec={1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
```

Vector Operations are also native:

```
spu_add(vec1,vec2);
spu_sub(vec2,vec1);
spu_mul(vec2,vec3);
```

I want to know if any of eight chars is greater than 15
How can it be done without a loop and expensive branch ? SPUs offer no branch prediction.

```
my_vector = spu_insert(my_vector,short_1,0);
my_vector = spu_insert(my_vector,short_2,1);
...
short_vector = spu_splats(15);
gather_vector = spu_cmpgt(my_vector,short_vector);
result_vector = spu_gather(gather_vector);
int value=spu_extract(result_vector,0);
bool our_bool = (bool)value;
```

Multi-Core Software Reengineering and Parallel Software Synthesis (Sketch)

Apollo Ellis, Rastislav Bodik UC Berkeley

“A Review of Real Time Pixel Based Ray Tracing Acceleration on Multi-core”

Apollo Ellis
University of California at Berkeley

Abstract

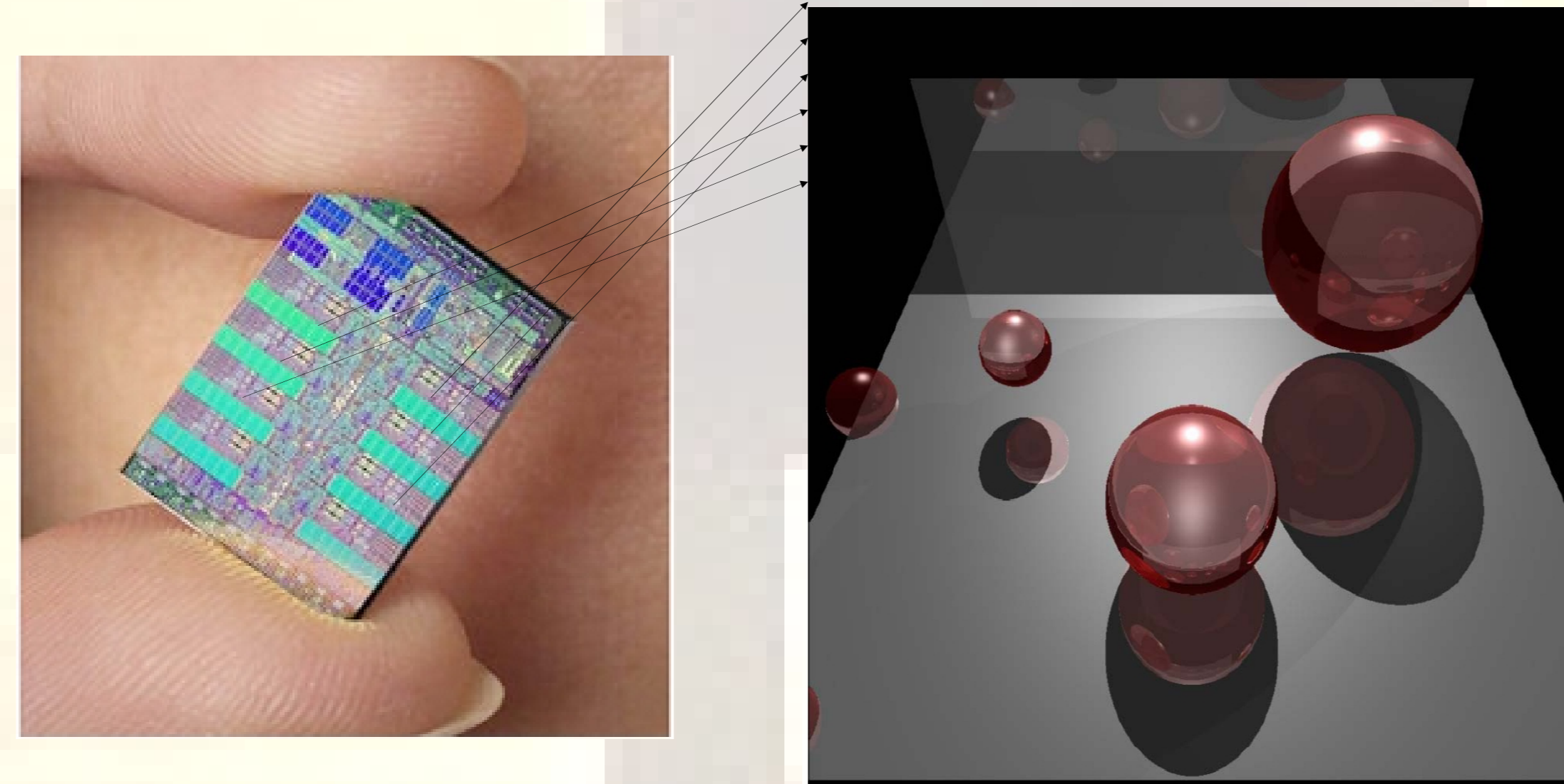
The term real time ray tracing could have in earlier years been referred to as somewhat of an oxymoron. Due to today's multi-core architectures and the parallel chips of the future the power and simplicity of ray tracing is coming into the grasp of the real time graphics application writer. In this paper I survey various acceleration techniques used in fast multi-core ray tracing. This paper is intended as a literature review and is concluded with considerations for design of an engineering effort in real time multi-core ray tracing. In terms of architecture this paper will be relative to shared and distributed memory multi-core processors with some mention of cluster systems. I will discuss various techniques used to overcome ray tracing inefficiency in the context of a somewhat standard approach to ray tracing. In particular I will examine four basic areas of ray tracing in which acceleration is possible and commonly attempted: programming model (i.e. demand driven parallel, data driven parallel) and parallel load balancing, instruction level hand tuning, ray-primitive intersection (simplified to triangle intersection), and space and object subdivision. In conclusion I will show the choices I am making and that were made in each area for techniques to use in optimization of multi-core ray tracing project.

Introduction

Ray tracing is a rendering algorithm that traces vectors through pixels on a virtual screen or camera lens. It tests their interaction with surfaces in a 3D graphic scene with respect to light intensity, reflection, refraction and shading. Historically the technique is widely unusable in real time applications because of its intense computation requirements. I simplify the basic algorithm to the following:

- Trace a ray through a screen pixel.
- Calculate it's intersection with objects in the scene or give the pixel the background color.
- If intersection occurs send a shadow ray from the hit point to each light source.
- Test the shadow ray for intersections on the way to the light source.
- If intersection occurs shade the hit point using the texture/material of the object and the ambient light
- Else shade hit point accordingly
- If there is reflection or refraction calculate their respective rays, recurse on the rays, and calculate their contribution on the hit point
- Return the final rgba value for the pixel

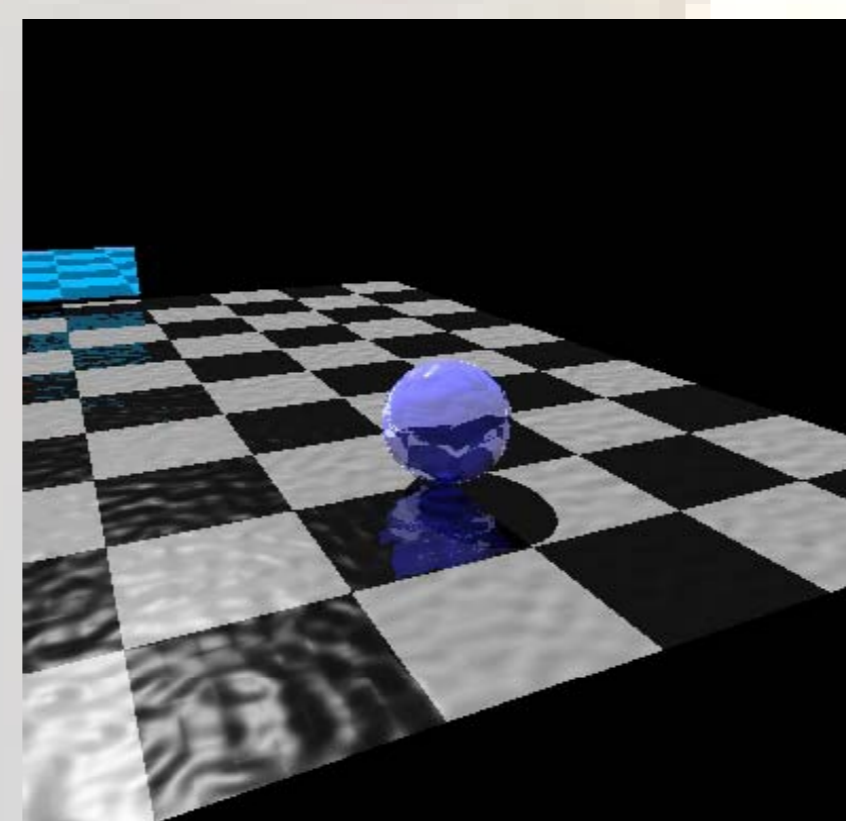
This is the workhorse algorithm in a basic ray tracer. It takes an order of magnitude longer to execute than any other part of ray tracing. To optimize away the inefficiency here is the core of this research.



MIT Blue Steel Ray Tracer

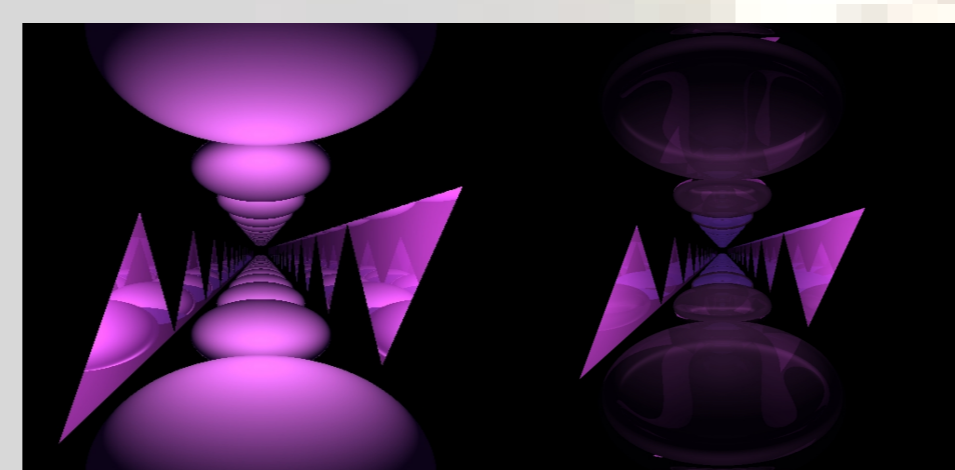
Assigns each core one row of pixels

On small scale scenes with few primitives: 15 or less
Rendering is fairly fast
Blue Steel Pong was created
On slightly larger scale scenes: 50 to a 100 primitives
Blue Steel rendering can't keep up with dynamic demands
Roughly 5 fps or less
Similar to single threaded ray tracing



Ray Tracing with The BSP Tree

Cut the space with a single plane, this is our root node.
Each side of the plane is a child node cut with it's own plane
Recurse down until we get back to a few primitives.
Store the structure and traverse it with rays we need traced.
Now it is possible to reduce the work done on a larger scene down to that which Blue Steel can work well with.



Note: With Fast Ray-Triangle Intersection algorithms such as Badouel's or Ingo Wald's and Large Size Ray Packet Tracing 6X speed ups can and have been achieved in addition to the BSP acceleration.

PS3DOOM

We wish to recreate on a multi-core platform the classic Doom game engine created at Id Software. Lead programmer John Carmack leaves us with some significant challenges in achieving a fast software based ray casting engine on the order of Doom.

The core technologies at our grasp came through rigorous exploration of the infamous Linux Doom source release by John Carmack and are centered around texturing and rendering of walls, floors and ceiling, and game sprites.

On the Playstation 3 every instruction possible in our implementation will be SIMD hopefully allowing us to achieve a 3 to 4 time speed up on most codes.

Due to the Visplane technologies we must turn to a sector based distribution of work amongst the multiple cores of the Cell. This should yield another 4 to 5 time speed up.

This is not a source port. This is my code.

Drawing walls default

Textures must be at least 128 pixels high.
TopPixels/bottomPixel = 384 + (floorheight/ceilingheight - viewz)*886/distance for (all wall hits)

```
Find the distance from the start vertex of seg to the hit point + offset.
Divide by the textureOffset. Subtract this multiplied by the textureOffset
from the distance. Multiply this by textureWidth and divide by
textureOffset to get the columnIndex.

count=topPixel-bottomPixel

do{
    Index into the texture as [column][i*frac >> 16]&127]
    Draw into the framebuffer
    i++;
}while(i<count)
```

HERE IT IS THE OBNOXIOUS VISPLANE (It basically just draws the floor :-)
As a ray traverses a scene it should record at each seg the bottom and top horizontal pixel number for a visplane of that seg's sector. Keep these planes in a list. Keep this list in a list and begin a new list anytime the order of sectors cross and thus of visplanes crossed changes. Render the visplanes from left to right down each list in turn.

```
PPU:
Baseyscale = finescos[(viewangle-ANG90)>>3]]
Baseyscale = finesine[... ]

for(i=0;i<viewheight<i++)
pixelsAcrossMultiplier[i] = (viewz-floorheight)/abs(i-viewheight/2)
pixelsAcrossMultiplier [i] |=1
```

For a span at height y

```
xStep = FixedMul(baseyscale , pixelsAcrossMultiplier [y])
yStep = FixedMul(baseyscale , pixelsAcrossMultiplier [y])
```

```
*****
width = pixelsAcrossMultiplier [y]*abs(x1-centerx);
length = FixedDiv (width , finesine[xtoviewangle[x1]>>3])
```

```
angle = viewangle + xtoviewangle >> 3
x = viewx + FixedMul(fincosine[angle],length)
y = viewy + FixedMul(finsine[angle],length)
*****
drawspan()
{
    while(i<count){
        color = texture[x&63][y&63]
        framebuffer[blah]=color
        x+=xstep
        y+=ystep
    }
}
```

SKETCH

SKETCH: Programming With Partial Programs
Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia UC Berkeley Vijay Saraswat IBM

Motivation
Programmers often face a difficult tradeoff between simplicity and efficiency
-Programmers would like to write clean and readable code, but such code usually has poor performance
-On the other hand, efficient code is difficult to write and maintain
With sketching, the programmer can write the clean and readable version of the program, and then provide only an outline of the efficient implementation

Example: Population Count
1) Write a clean reference implementation
For this example we want to compute the number of ones in an input word. For this task, it's easy to write a simple specification for it, but it's going to be very inefficient.
let pop (bit[W]) x;
let count = 0;
for (let i = 0; i < W; i++) if (x[i] counts++) ;
return count;

2) Think of a clever optimization trick
We can solve this in a divide-and-conquer fashion, computing all the sums at each level in parallel.

3) Write a sketch using the sketching constructs
The implementation idea can be described as a sketch. Each hole in the sketch will be filled out with the correct code. The implements keyword denotes functional equivalence.

4) The resolved sketch is compiled into C code
unassigned positions remain unassigned so I
x = (x & 0x5555) + (x >> 1) & 0x5555;
x = (x & 0x3333) + (x >> 2) & 0x3333;
x = (x & 0x00FF) + (x >> 8) & 0x00FF;
return x;

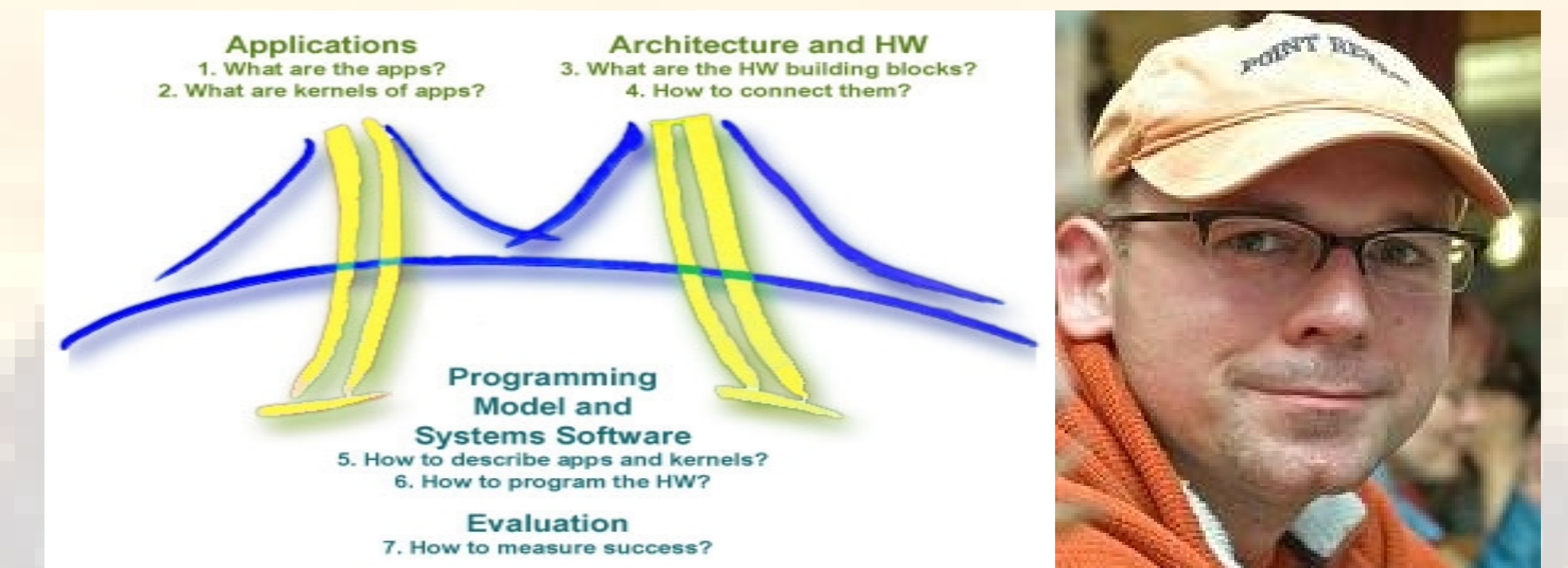
Example: Merge Sort
This sketch for Merge Sort shows that some holes can be arbitrary expressions, rather than just constants.

System Architecture
1) Write a clean reference implementation.
2) Think of a clever optimization trick.
3) Write a sketch using ?? constructs.
4) Indicate functional equivalence.
5) Compiler tries to assign values to ?? in order to satisfy the equivalence.
6) If successful == C code

Solver Algorithm
The sketch synthesis problem is an instance of ZQBF:
∃ x ∈ {0,1}^n, ∀ y ∈ {0,1}^m, F(x) = S(x, y)
Counter-example driven solver:
- Reduces ZQBF to a series of SAT problems
synthesizeForSomeInputs()
- finds a set of concrete values that the sketch produces the same output as the spec for all inputs in I
- establishes partial equivalence
verifyForAllInputs()
- Given a concrete, find an input y for which the sketch output differs from the spec output
- is a concrete example of equivalence

RELATED WORK AND RESULTS

Sketching Goes Concurrent: Parallel Data Structures Paper Accepted at PLDI '08



MS Intel Par Lab For UCB Parallel Research !!

We See It Like This



Parallel Browser on iPhone



Web 3.0 at 3.0 Watts

Credits

Ras Bodik: Advisor

Armonda Solar-Lezama and Gilad Arnold: Sketch Research

Chris Jones: Mentor
Parsing Web 3.0 with 3.0 Watts.

MIT: Cell Pics & Blue Steel

John Carmack: Obnoxious Visplanes