

Parallelism through Circuit Design

John O'Donnell
University of Glasgow

Programming Models for Ubiquitous Parallelism
Dagstuhl Seminar
September 4, 2007

How to make computers faster?

Marketing people used to talk all the time about clock speed

That's because the clock speed kept getting better

Now they don't mention it so often.

What happened?

Improving clock speed

Only a few ways to do it

- The physicists give you faster transistors
- You reduce the critical path depth or wire length via clever design

But

- It's getting harder to speed up the transistors
- The critical path depth can't get much lower
- Wires are not getting shorter

Even worse...

- In synchronous circuits, the large number of transistors causes overhead that slows the clock
- And don't yet know how to design very large asynchronous circuits effectively

How else to make computers faster?

- Get more work done in each clock cycle, so you don't need so many
- This is where clever architecture comes in
- Typically, a clever idea that brings speed requires more transistors to implement it

How many transistors do you need?

- You can design a CPU with ~1,000 transistors. It's Turing Complete, but slow.
- With more transistors, you can make it faster
 - Increase the word size
 - Replace ripple adder with carry lookahead
 - Cache
 - Pipelining
 - Superscalar

There is little relationship between the cost in transistors and the speedup you get!

Diminishing returns

- These techniques can be used one time only
 - Once the processor is pipelined, you can't pipeline it again for another speedup
- The most profligate use of transistors is for overcoming misfeatures in the instruction set

The Program Complexity Game

- You're given a problem to solve.
- You're given a **programming language**, and can **combine its constructs** any (legal) way you like.

Rule breakers get error messages

The goal:

- Find a correct solution with the best speed you can attain.
- Speed measured in some suitable model, maybe **$O(n \log n)$ instructions executed**

The Circuit Complexity Game

- You're given a problem to solve
- You're given a box of **logic gates and flip flops**, and can **wire them together** any (legal) way you like

*Rule breakers
get blown fuses*

The goal:

- Find a correct solution with the best speed you can attain
- Speed measured in some suitable model, maybe **$O(\log n)$ clock cycles with critical path depth of 47**

How Can You Win?

You could

- Hire somebody to build you some CPUs
- Then play the Program game

But is that really the best you can do?

Sometimes Special Circuits Win

- Special functional units
 - If you have only software floating point, it's better to get hardware floating point than to go for minor code optimisations
- If you have a data parallel algorithm, you may do better with true SIMD parallelism (straightforward as a circuit) rather than multicore

A Conjecture

- The von Neumann architecture is not optimal for some problems
- In other words:
 - If you're playing the circuit game, and you used all your transistors to build a multicore, *you have already lost*

A Piece of Evidence

There's a data structure called a **functional array**, with operations *update*, *lookup*, *release*

- A serious conjecture: there does not exist an implementation (on von Neumann architecture) that always gets constant time for all these operations
- But with circuit parallelism, you can do it!

(Functional arrays are not just arrays in a functional program; it's a different data structure. Discovered by functional programmers, but it's independent of the language.)

How does it work?

- The solution is a data parallel algorithm
- Each operation (lookup, update, release) requires a little bit of computation on *every word in the memory*
- Terrible for multicore!
- But easy for a circuit—it takes just a few transistors to add a little computational ability to each word

Hard data parallelism

- This isn't just parallelism organised around the data
- It's more like SIMD parallelism, where you compute on every data element simultaneously

Task and Data Parallelism

- These are programming models
- Data parallelism fits well on SIMD, but now it's usually used on MIMD architectures
- Each model can be implemented on top of the other
- Many data parallel applications run fine on multicore...

but some of them really need circuit parallelism

Is it practical to design circuits?

- It takes a long time to get an ASIC fabricated

Application-specific Integrated Circuit, a chip for an application

- But now you can do almost as well with FPGAs

Field Programmable Gate Array

- You just program them, and they behave like your circuit design

What is an FPGA?

- An array of cells containing
 - General logic boxes
 - Flip flops
 - A programmable wiring network
 - Some control memory that
 - Tells the logic boxes what function to compute
 - Tells the wiring network which connections to make

Similar to MPP, a SIMD architecture

FPGAs vary a lot

Some FPGAS also have

- Adders
- Memories
- Efficient processor cores

Metaphors

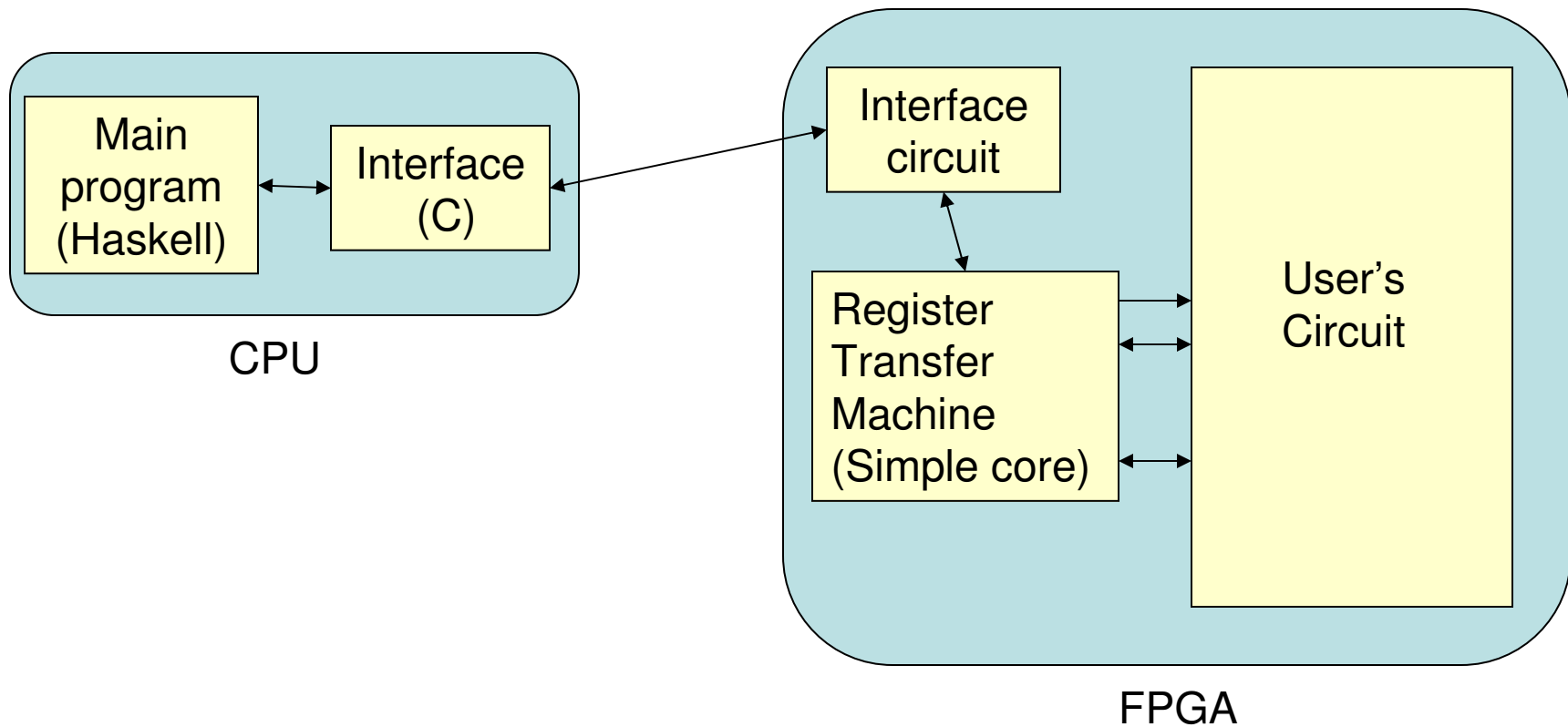
- Some people call this “soft hardware”

Nevertheless, the hardware on an FPGA is really fixed, and you’re just programming a general purpose device

Are FPGAs too hard for programmers?

- The skills needed are circuit design, not ordinary programming
- The tools usually used are complicated and inexpressive

A support framework



Programming languages...

Functional languages give programmer good level of abstraction

- It's high level, and far from the instruction set

```
load R2,$0a34[R2]
add  R5,R2,R8
```

....

Machine language
Commands

```
h = scanl f a . map g
```

Functional programming
Equations

Imperative languages closer to instructions, but not as good at abstraction

Hardware description languages

- Industry standard VHDL

- Claimed to be high level

- Based essentially on assignments, including parallel assignments

- But circuits don't behave like assignments, they behave like functions

VHDL = VHSIC Hardware Description Language

VHSIC = Very High Speed Integrated Circuit



Parallelism through Circuit Design

A key discovery

- Combinational (stateless) circuits are obviously functions
- Sequential (stateful) circuits are also **pure functions over streams** (Steve Johnson, 1981)

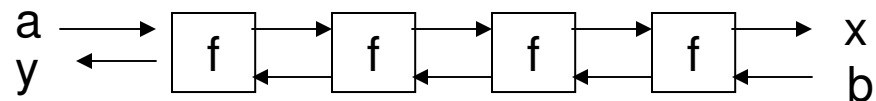
Functional hardware description

- A pure nonstrict (lazy) functional language (like Haskell) is ideal for hardware description
 - All the usual advantages still hold (excellent abstraction, higher order functions, types, ...)
 - And it corresponds with real hardware much better than imperative languages

Hydra

- A domain-specific language for hardware description, based on Haskell
- Not too difficult to learn
 - An elementary class worked from logic gates up to a simple CPU in eight lectures
 - There's a systematic design methodology
 - Some problems seem *easier* to solve with circuits than with programs

Wiring components together



You can give each signal explicitly:

Shifter $a\ b = (x,y)$

where ...

$(x_2,y_2) = f\ x_1\ y_3$

....

Or you can capture the pattern with a higher order function:

Shifter = `bscan f`

The big picture

- Programmer partitions algorithm into part to run on CPU, and part to run as circuit
 - No automatic support for this
 - But, you can write both parts in Haskell, refactor the program, reason about correctness, etc.
- Compilation
 - ghc translates software part into machine language
 - Hydra translates hardware part onto FPGA

A practical problem

- Vendors of FPGAs don't give you
 - Details of their architecture
 - The ability to program the logic boxes and wiring networks
- Instead, they
 - “Let” you program in VHDL
 - Provide tools that do synthesis, placement and routing automatically

Problems with VHDL

The Hydra circuit description

- Benefits hugely from higher order functions
VHDL doesn't have them

- Generates a precise netlist

Why hide it, and make VHDL generate a worse one?

- Knows about locality, useful for deciding where to put everything on the chip

Using VHDL, you throw this information away, and the automatic tools cannot get it back

VHDL is said to be higher level than Hydra, but 7 lines of Hydra translated into ~200 lines of VHDL

Exotic circuits

An asynchronous 3SAT solver

Joint work with P. Cockshott, A. Koltes, P. Prosser

Motivation: work by Kaufmann on stabilisation of Boolean networks

An array:

- Vertical signals carry guesses for variable settings
- Horizontal signals give evaluation of a term
- More vertical signals propagate failure, and boxes decide randomly whether to change a signal when a term is failing

Initial experiments

- Prototype implementation on FPGA
- Compared with state of the art software solver, considering random satisfiable instances
- For problem instances that aren't close to the transition boundary, significantly faster than a state of the art software solver

You too can design circuits!

You just need

- To learn about it the right way
- Effective ways to think about hardware
- Suitable language for describing the circuit, simulating it, and fabricating it
- Suitable fabrication (FPGAs)

But when is it appropriate?

- “Hard” data parallelism
- Nonstandard functional units
- Exotic techniques, like asynchronous feedback
- Whatever new ideas you think of

Conclusion

- Multicore is valuable, but not the only way to make use of more transistors!
- Circuit design is not just for the computer designers
 - It isn't that difficult
 - It can give outstanding performance
- But you need
 - To use it when it's appropriate
 - Suitable tools