Non Von Neumann Computation
(a survey)

Eric Schulte

University of New Mexico

03 May 2010

Outline

1. Background
   - Von Neumann Architecture
   - Backus calls for Non Von Neumann Computation
   - Moores Law
2. Non Von Neumann Architectures, Past and Present
   - Reduction Machines
   - Message Passing
   - Data-Flow / Stream-Processing
   - Functional Programming Languages
3. Conclusion
   - Conclusion
4. Bibliography
   - Bibliography

Outline

1. Background
   - Von Neumann Architecture
   - Backus calls for Non Von Neumann Computation
   - Moores Law
2. Non Von Neumann Architectures, Past and Present
   - Reduction Machines
   - Message Passing
   - Data-Flow / Stream-Processing
   - Functional Programming Languages
3. Conclusion
   - Conclusion
4. Bibliography
   - Bibliography

Von Neumann's Preliminary Discussion [Burks et al., 1946]

"Inasmuch as the completed device will be a general-purpose computing machine it should contain certain main organs relating to arithmetic, memory-storage, control and connection with the human operator. It is intended that the machine be fully automatic in character, i.e. independent of the human operator after the computation starts."

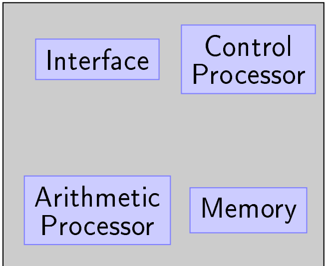


Figure: four main organs of computation

Von Neumann Architecture and Languages

Backus's "Bottleneck" [Backus, 1978]

Von Neumann Architecture and Languages

Backus's "Bottleneck" [Backus, 1978]

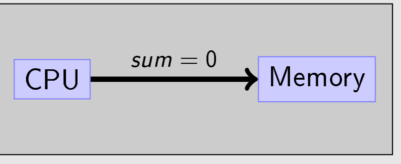
CPU — Von Neumann Bottleneck — Memory

Von Neumann (Imperative) Languages

```
1 int sum = 0;
2 for(int i=0;i<COL_SIZE;i++)
3   sum = sum + col[i];
```

Von Neumann Architecture and Languages

Backus's "Bottleneck" [Backus, 1978]

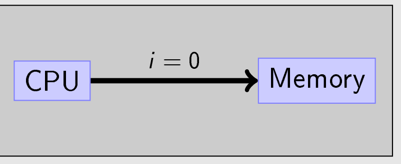


Von Neumann (Imperative) Languages

```
1 int sum = 0;
2 for(int i=0;i<COL_SIZE;i++)
3   sum = sum + col[i];
```

Von Neumann Architecture and Languages

Backus's "Bottleneck" [Backus, 1978]



Von Neumann (Imperative) Languages

```
1 int sum = 0;
2 for(int i=0;i<COL_SIZE;i++)
3   sum = sum + col[i];
```

Von Neumann Architecture and Languages

Backus's "Bottleneck" [Backus, 1978]

CPU → Memory (&col[i])

Von Neumann (Imperative) Languages

```
1 int sum = 0;
2 for(int i=0;i<COL_SIZE;i++)
3   sum = sum + col[i];
```

Von Neumann Architecture and Languages

Backus's "Bottleneck" [Backus, 1978]

CPU ← $\overline{col}[i]$ ← Memory

Von Neumann (Imperative) Languages

```
1 int sum = 0;
2 for(int i=0;i<COL_SIZE;i++)
3   sum = sum + col[i];
```

Von Neumann Architecture and Languages

Backus's "Bottleneck" [Backus, 1978]

Von Neumann (Imperative) Languages

```
1 int sum = 0;
2 for(int i=0;i<COL_SIZE;i++)
3   sum = sum + col[i];
```

Von Neumann Architecture and Languages

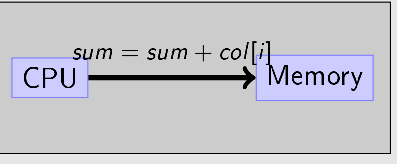
Backus's "Bottleneck" [Backus, 1978]

Von Neumann (Imperative) Languages

```
1 int sum = 0;
2 for(int i=0;i<COL_SIZE;i++)
3   sum = sum + col[i];
```

Von Neumann Architecture and Languages

Backus's "Bottleneck" [Backus, 1978]



Von Neumann (Imperative) Languages

```
1 int sum = 0;
2 for(int i=0;i<COL_SIZE;i++)
3   sum = sum + col[i];
```

Von Neumann Architecture and Languages

Backus's "Bottleneck" [Backus, 1978]

Von Neumann (Imperative) Languages

```
1 int sum = 0;
2 for(int i=0;i<COL_SIZE;i++)
3   sum = sum + col[i];
```

Von Neumann Architecture and Languages

Backus's "Bottleneck" [Backus, 1978]

Von Neumann (Imperative) Languages

```
1 int sum = 0;
2 for(int i=0;i<COL_SIZE;i++)
3   sum = sum + col[i];
```

Von Neumann Architecture and Languages

Backus's "Bottleneck" [Backus, 1978]

Von Neumann (Imperative) Languages

```
1 int sum = 0;
2 for(int i=0;i<COL_SIZE;i++)
3   sum = sum + col[i];
```

Von Neumann Architecture and Languages

Backus's "Bottleneck" [Backus, 1978]

Von Neumann (Imperative) Languages

```
1 int sum = 0;
2 for(int i=0;i<COL_SIZE;i++)
3   sum = sum + col[i];
```

Von Neumann Architecture and Languages

Backus's "Bottleneck" [Backus, 1978]

Von Neumann (Imperative) Languages

```
1  int sum = 0;
2  for(int i=0;i<COL_SIZE;i++)
3    sum = sum + col[i];
```

Can programming be liberated from the Von Neumann style [Backus, 1978]

Architectures

- new organizations of Processor and Memory
- eliminate "Von Neumann Bottleneck"

Languages

- declarative – static and non-repetitive
- point free – no named variables
- polymorphic – applicable to multiple type
- amenable to mathematical analysis
  - functions built from a set of *primitive functions* through the application of higher order *functional forms*
  - All functions are $\bot$ preserving, $\forall f,\ f : \bot = \bot$

Can programming be liberated from the Von Neumann style [Backus, 1978]

leads to much excitement, and a flurry of activity

- new work in functional programming languages
- new work in non Von Neumann architectures

this work was *mostly doomed*

Moores Law

The number of transistors which can placed on a chip doubles roughly every two years.

Limits to Moore's Law

In terms of size [of transistors] you can see that we're approaching the size of atoms which is a fundamental barrier, but it'll be two or three generations before we get that far but that's as far out as we've ever been able to see. We have another 10 to 20 years before we reach a fundamental limit. By then they'll be able to make bigger chips and have transistor budgets in the billions. – Gordon Moore.

Transistor ≠ Speed

- clockspeed ceiling at 3 Ghz
- energy dissipation
- non processor bottlenecks
- majority of chip space devoted to cache

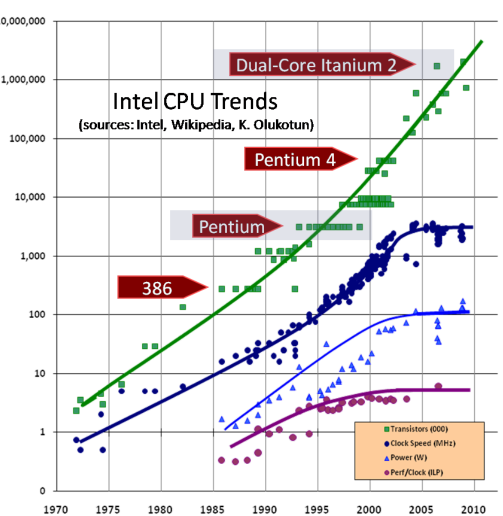


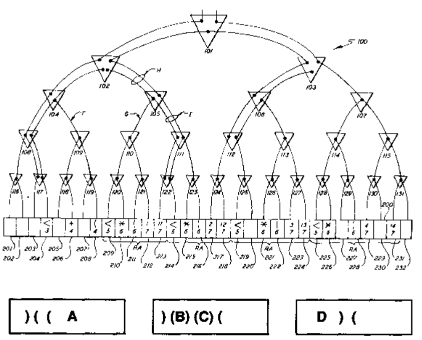
Figure: Intel CPU Trends [Sutter, 2005]

Outline

1. Background
   - Von Neumann Architecture
   - Backus calls for Non Von Neumann Computation
   - Moores Law
2. Non Von Neumann Architectures, Past and Present
   - Reduction Machines
   - Message Passing
   - Data-Flow / Stream-Processing
   - Functional Programming Languages
3. Conclusion
   - Conclusion
4. Bibliography
   - Bibliography

Cellular Tree Architecture [Treleaven and Mole, 1980]

- massively parallel
- Used to execute FFP (described in detail later) and λ-calculus



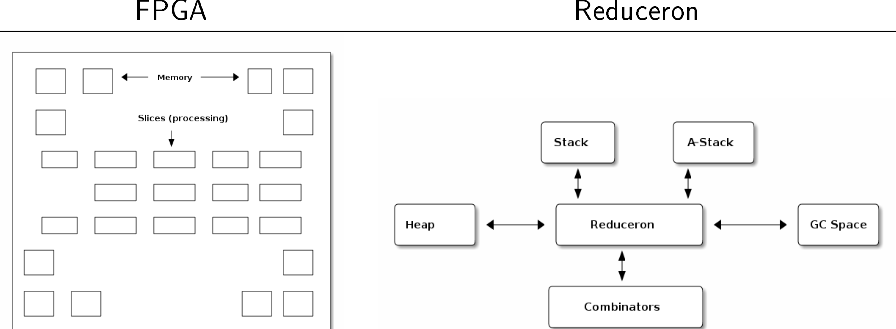
Reduceron [Naylor and Runciman, 2007]

Graph reduction implemented on an FPGA

TI – Template Instantiation

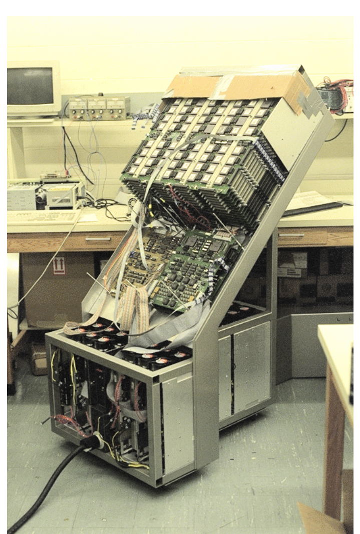
- memories – can all be accessed in parallel
- Haskell $\to$ YHC $\to$ TI (λ-calculus)

FPGA	Reduceron



"Jelly Bean" Machine [Spertus et al., 1993]

- messaging between many CPUs
- memory is "on chip"
- "processors are cheap"
- "memory is expensive"
- ran a version of *smalltalk*



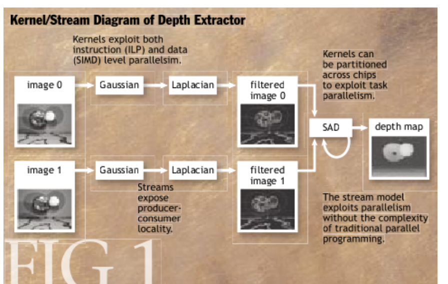
Erlang [Armstrong, 2007]

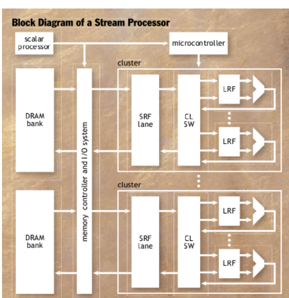
- originally created by *Ericsson* for telephony applications, later open sourced
- concurrent programming language and runtime system
- functional language with message passing primitives for IPC
- currently gaining popularity, used in both *Facebook* and *Amazon*

Figure: Erlang logo

Stream Processor [Dally et al., 2004]

- Actually used in modern systems, esp. for high computation/watt ratio
- *Merrimac* stream processing super-computer under development at Stanford [?]





Propagators [Sussman and Radul, 2009]

components

propagators functions which connect input cells to output cells, the execution of which is triggered when the value of an input cell is altered

cells local data stores the contents of which are get and set by propagators

example propagator program ($\sqrt{x}$)


```
1  (defun heron [x done guess]
2    (if done
3          guess
4        (/ (+ guess
5               (/ x guess)) 2.0)))
6
7  (defun done? [x guess] [done]
8    (if (< (abs
9              (- (* guess guess) x))
10            0.001)
11          true
12        false))
```

Backus's FP/FFP [Backus, 1978]

FP

1. a set O of Objects – an *atom* x or a sequence of *atoms* $< x_1, x_2, \ldots >$
2. a set F of *functions*, $f : O \rightarrow O$
3. function *application*
4. a set $\mathbb{F}$ of *functional forms*
5. a set D of *definitions*

FFP

In FFP systems objects are used to "represent" functions in a systematic way. Otherwise FFP systems mirror FP systems closely.

Representation μ of an expression returns the Object which is its *meaning*, and ρ of an Object returns the function which is represents.

Cells allow for state in FFP systems, for the storing of both defined functions and objects.

APL – "A Programming Language" [Falkoff and Iverson, 1973]

- Array processing language, with *Exotic syntax* and *Concise (Obfuscated) Programs*
- Manipulates entire arrays atomically
- Still in active use today
- Combined with FP to create the *J* language

```
⍎'⎕',∊N⍴⊂S←'←⎕←(3=T)∨M∧2=T←⊃+/(V⌽"⊂M),(V⊖"⊂M),(V,⌽V)⌽"(V,V←1 ¯1)⊖"⊂M'
```

Clojure [Halloway, 2009]

- dialect of lisp
- run on the Java Virtual Machine (JVM)
- functional language
- concurrent language
  - all data is immutable, unless wrapped in synchronization constructs
  - software transactional memory system
  - agent system

Figure: Clojure logo

Haskell [Hudak et al., 2007]

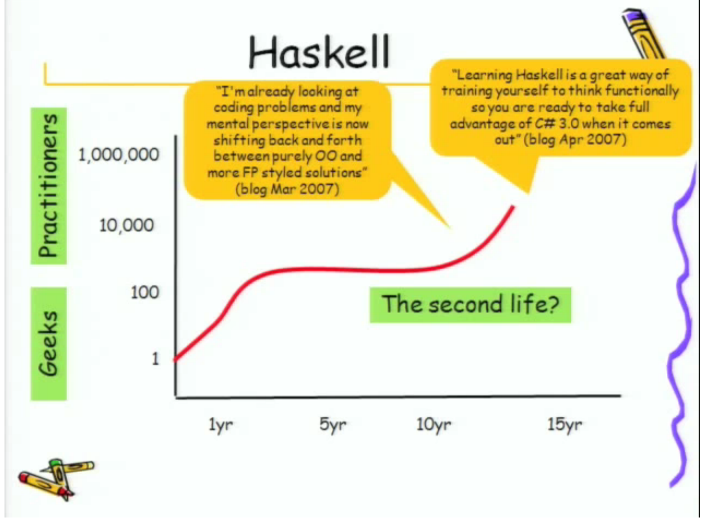


Figure: Haskell Popularity

Outline

1. Background
   - Von Neumann Architecture
   - Backus calls for Non Von Neumann Computation
   - Moores Law
2. Non Von Neumann Architectures, Past and Present
   - Reduction Machines
   - Message Passing
   - Data-Flow / Stream-Processing
   - Functional Programming Languages
3. Conclusion
   - Conclusion
4. Bibliography
   - Bibliography

Conclusion

Given that;

- the speed of serial processors are no longer increasing
- nearly all new processors are multi-core
- VN-bottleneck has become the limiting factor of computer performance, and leading cause of energy consumption

computer programmers and system architects are turning to *non Von Neumann* models of computation running on

- Traditional Von Neumann machines
- Networked Von Neumann machines
- Virtual Machines
- non-Von Neumann hardware

Outline

1 Background
- Von Neumann Architecture
- Backus calls for Non Von Neumann Computation
- Moores Law

2 Non Von Neumann Architectures, Past and Present
- Reduction Machines
- Message Passing
- Data-Flow / Stream-Processing
- Functional Programming Languages

3 Conclusion
- Conclusion

4 Bibliography
- Bibliography

Bibliography I

Armstrong, J. (2007).

A history of erlang.

In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 6–1–6–26, New York, NY, USA. ACM.

Backus, J. W. (1978).

Can programming be liberated from the von neumann style? a functional style and its algebra of programs.

Commun. ACM, 21(8):613–641.

Burks, A. W., Goldstine, H. H., and von Neumann, J. (1946).

Preliminary discussion of the logical design of an electronic computing instrument.

Technical report, Institute for Advanced Study, Princeton, NJ, USA.

Bibliography II

Dally, W. J., Kapasi, U. J., Khailany, B., and Ahn, J. H. (2004).
Stream processors: Programmability with efficiency.
ACM Queue, pages 52–62.

Falkoff, A. D. and Iverson, K. E. (1973).
The design of apl.
IBM Journal of Research and Development, 17(5):324–334.

Halloway, S. (2009).
Programming Clojure.
Pragmatic Bookshelf.

Hudak, P., Hughes, J., Jones, S. L. P., and Wadler, P. (2007).
A history of haskell: being lazy with class.
In *HOPL*, pages 1–55.
http://research.microsoft.com/en-us/um/people/simonpj/papers/history-of-haskell/.

Bibliography III

Naylor, M. and Runciman, C. (2007).

The reduceron: Widening the von neumann bottleneck for graph reduction using an fpga.

In *IFL*, volume 5083 of *Lecture Notes in Computer Science*, pages 129–146. Springer.

http://www.cs.york.ac.uk/fp/reduceron/.

Spertus, E., Goldstein, S. C., Schauser, K. E., von Eicken, T., Culler, D. E., and Dally, W. J. (1993).

Evaluation of Mechanisms for Fine-Grained Parallel Programs in the J-Machine and the CM-5.

In *Proceedings of the 20th International Symposium on Computer Architecture (ISCA)*, San Diego, CA.

Sussman, G. J. and Radul, A. (2009).

The art of the propagator.

CSAIL Technical Reports.

Bibliography IV

Sutter, H. (2005).
The free lunch is over: A fundamental turn toward concurrency in software.
Dr. Dobb's Journal, 30(3).
http://www.gotw.ca/publications/concurrency-ddj.htm.

Treleaven, P. C. and Mole, G. F. (1980).
A multi-processor reduction machine for user-defined reduction languages.
In *ISCA '80: Proceedings of the 7th annual symposium on Computer Architecture*, pages 121–130, New York, NY, USA. ACM.