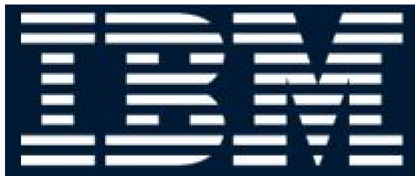
# Dataflow: Best of Times or Worst of Times?

**DFM'2019:**
**Dataflow Model Workshop 2019**

Erik Altman


Research

© 2019 IBM Corporation

# Talk Theme

It was the best of times, it was the worst of times.

*Charles Dickens, "A Tale of Two Cities"*

© 2019 IBM Corporation

# Caveats

- It has been many years since I have devoted significant research effort to dataflow models
- Views here are of a fan looking from a distance with nostalgia and hope

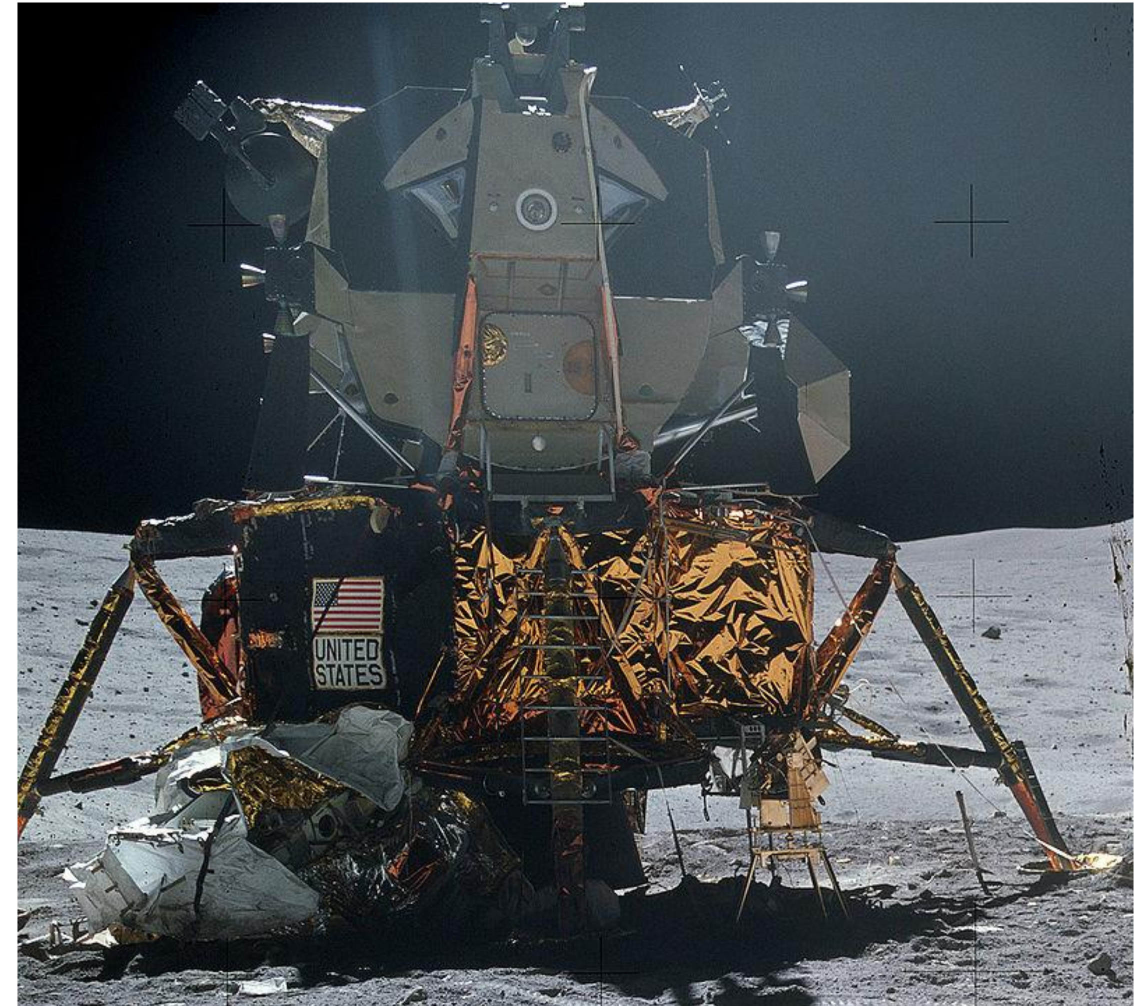
© 2019 IBM Corporation

# Aside

**July 19, 2019**

50 years less a day from the first moon landing

*If we can put a person on the moon, can we make dataflow succeed?*



© 2019 IBM Corporation

# Dataflow Popularity

## Citations over Time

- ISCA’1975: Jack B. Dennis and David P. Misunas
- *“A preliminary architecture for a basic data-flow processor”*
- https://dl.acm.org/citation.cfm?id=642111

16
14
12
10
8
6
4
2
0

1975 1977 1979 1981 1983 1985 1987 1989 1991 1993 1995 1997 1999 2001 2003 2005 2007 2009 2011 2013 2015 2017

© 2019 IBM Corporation

# Dataflow Popularity

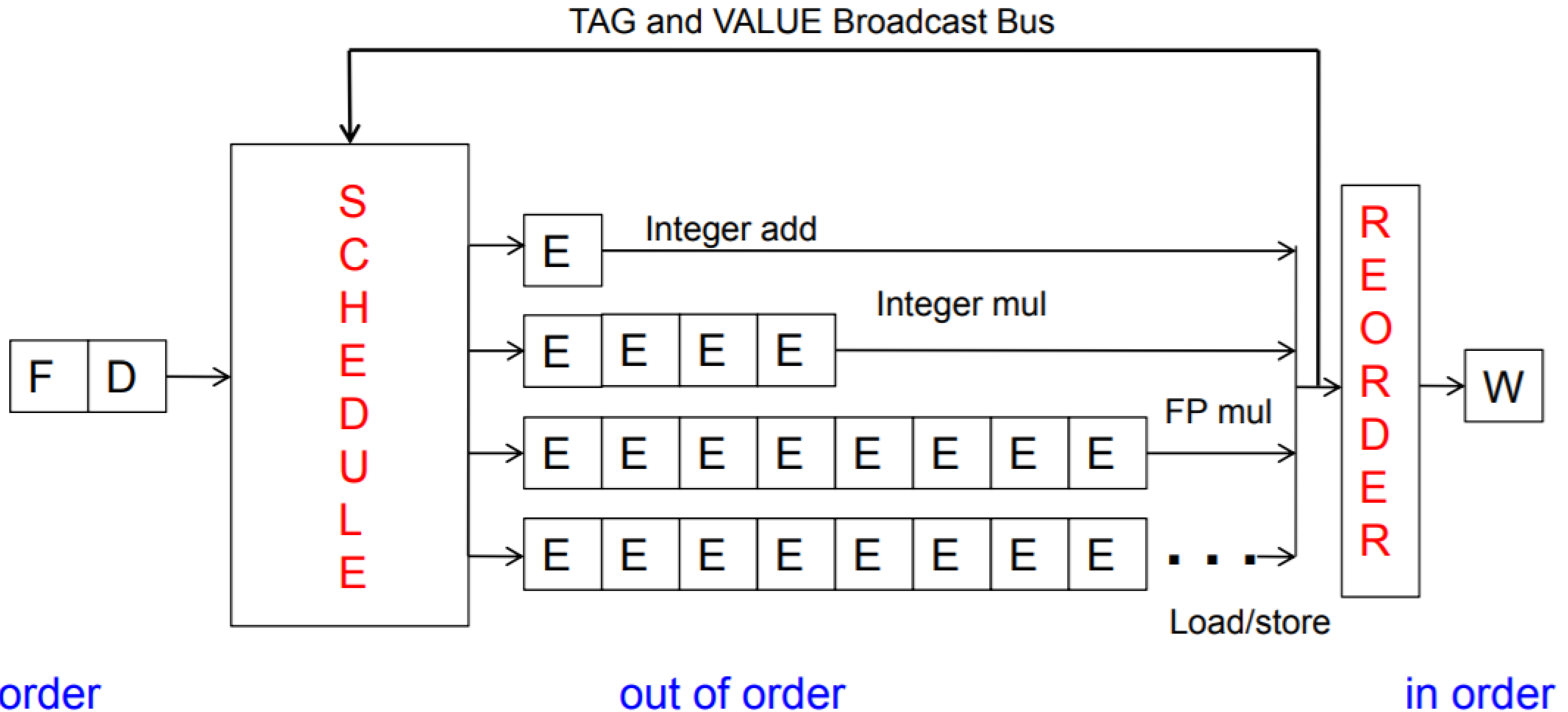
## Citations over Time

- ISCA'1975: Jack B. Dennis and David P. Misunas
- *"A preliminary architecture for a basic data-flow processor"*
- https://dl.acm.org/citation.cfm?id=642111

| Year | Citations |
|---|---|
| 1975 | 3 |
| 1976 | 1 |
| 1977 | 1 |
| 1978 | 1 |
| 1979 | 15 |
| 1980 | 7 |
| 1981 | 10 |
| 1982 | 8 |
| 1983 | 2 |
| 1984 | 2 |
| 1985 | 8 |
| 1986 | 3 |
| 1987 | 3 |
| 1988 | 2 |
| 1989 | 2 |
| 1990 | 1 |
| 1991 | 1 |
| 1992 | 1 |
| 1993 | 1 |
| 1994 | 1 |

© 2019 IBM Corporation

# Early Dataflow Success: Tomasulo / Reorder Buffers




*Onur Mutlu*
https://www.archive.ece.cmu.edu/~ece740/f11/lib/exe/fetch.php?media=wiki:lectures:onur-740-fall11-lecture10-ooo-afterlecture.pdf



© 2019 IBM Corporation

# # of Processors in Top500 Systems

IBM

## # of Cores

100,000,000
10,000,000
1,000,000
100,000
10,000
1,000
100
10
1

Max # of Cores

Min # of Cores

Jun-93 Jul-94 Aug-95 Sep-96 Oct-97 Nov-98 Dec-99 Jan-01 Feb-02 Mar-03 Apr-04 May-05 Jun-06 Jul-07 Aug-08 Sep-09 Oct-10 Nov-11 Dec-12 Jan-14 Feb-15 Mar-16 Apr-17 May-18 Jun-19

*Why has dataflow not emerged as the dominant paradigm for Top500 workloads?*

© 2019 IBM Corporation

# Successful Software Transitions

- **Spreadsheets:** Visicalc → Lotus 123 → Microsoft Excel
- **Word Processors:** Wang → Wordperfect → Word
- **Browsers:** Netscape / Mozilla → Internet Explorer → Chrome / Safari


© 2019 IBM Corporation

# Language Popularity

| May 2019 | May 2018 | Change | Programming Language | Ratings | Change |
|---|---|---|---|---|---|
| 1 | 1 | | Java | 16.0% | -0.38% |
| 2 | 2 | | C | 14.2% | +0.24% |
| 3 | 3 | | C++ | 8.1% | +0.43% |
| 4 | 4 | | Python | 7.8% | +2.64% |
| 5 | 6 | ^ | Visual Basic .NET | 5.2% | +1.07% |
| 6 | 5 | v | C# | 4.0% | -0.42% |
| 7 | 8 | ^ | Javascript | 2.7% | -0.23% |
| 8 | 9 | ^ | SQL | 2.6% | +0.57% |
| 9 | 7 | v | PHP | 2.5% | -0.83% |
| 10 | 13 | ^ | Assembly Language | 1.8% | +0.82% |
| 11 | 15 | ^^ | Objective-C | 1.6% | -0.69% |
| 12 | 12 | | Delphi/Object Pascal | 1.4% | +0.39% |
| 13 | 18 | ^^ | Perl | 1.4% | -0.48% |
| 14 | 16 | ^ | MATLAB | 1.4% | -0.44% |
| 15 | 10 | vv | Ruby | 1.3% | -0.16% |

C Flavors: **28%**
Fortran #27: **0.5%**
Lisp #32: **0.4%**
Haskell #45: **0.2%**

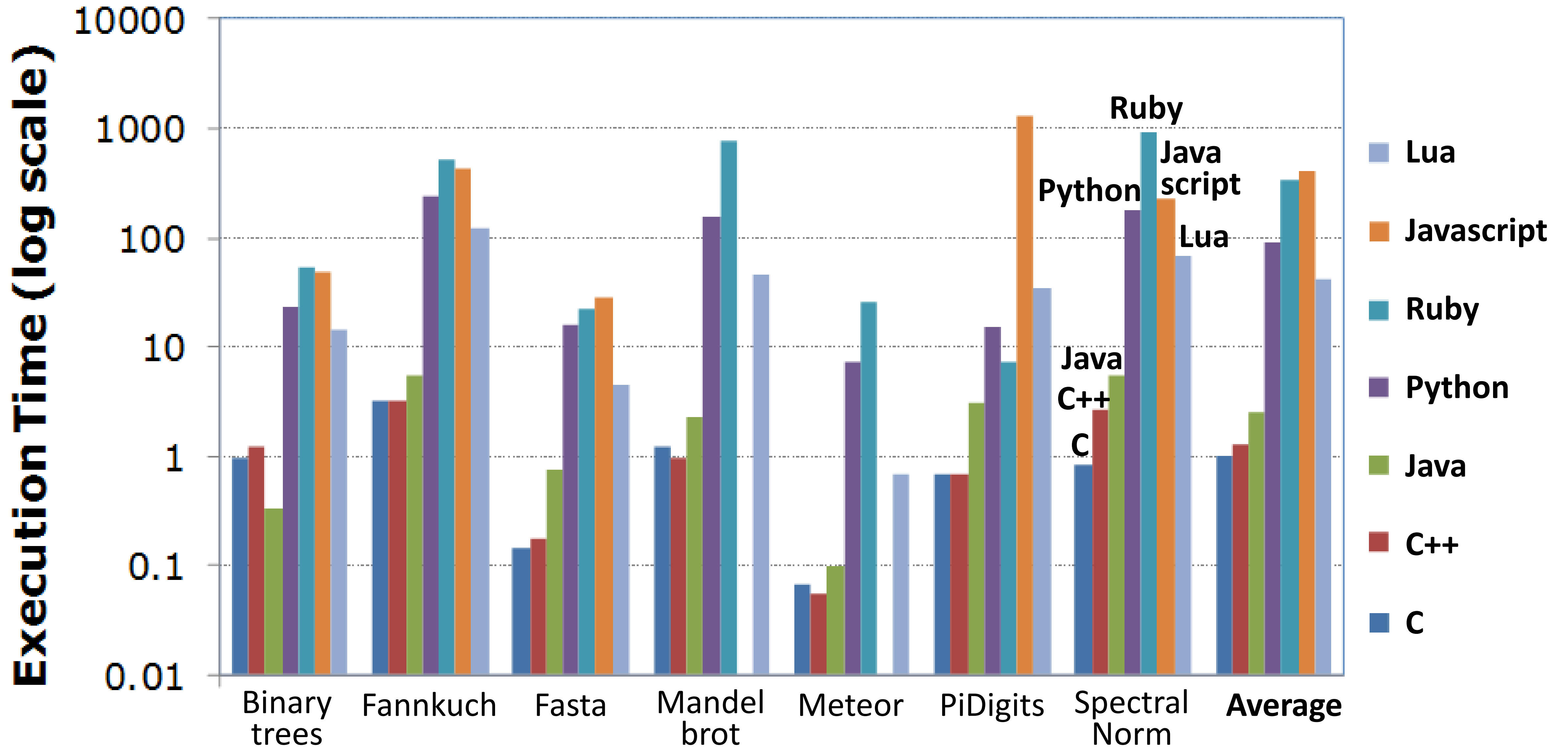
*No major niche where dataflow dominates!*

https://www.tiobe.com/tiobe-index

© 2019 IBM Corporation

# Do Programmers Care about Performance?

IBM




*D. Edelsohn, M. Gschwind, J. Moreira, P. Nagpurkar, M. Valluri*



© 2019 IBM Corporation

# The Challenge of Parallelism

"When we start talking about **parallelism** and ease of use of truly parallel computers, we're talking about a problem that's **as hard as any that computer science has faced**.

**I would be panicked if I were in industry**."

**John Hennessy**

Turing Laureate

CS Professor and President Emeritus, Stanford

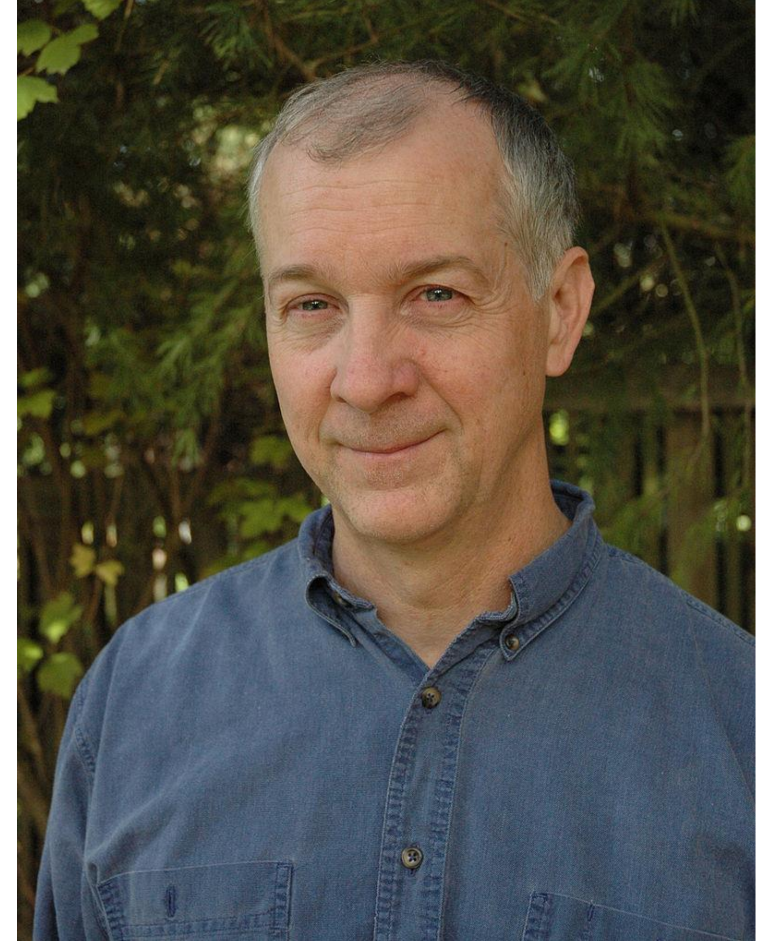
Author – Best-selling Computer Architecture textbook

Chair – Alphabet / Google

© 2019 IBM Corporation

# Popularity is Essential

- **“He who makes the most silicon wins”**
  - Bob Colwell



© 2019 IBM Corporation

# Killer Apps

- **Java:** JEE and WebSphere-like middleware
- **C:** Operating systems, Realtime, IoT
- **Python:** Deep Learning Frameworks
- **Fortran:** HPC
- **Lisp:** Symbolic AI
- **PCs:** Spreadsheets, Word Processing
- **Smartphones:** Texting, Pictures, Map Guidance, Social Networks, News

- *What is the dataflow killer app?*

© 2019 IBM Corporation

# Who is the target audience for Dataflow?

- All programmers
- HPC programmers
- DSP programmers
- Data scientists
- Deep learning algorithm developers
- Other domains

© 2019 IBM Corporation

# What tasks can dataflow make easier?

- **Exploiting parallelism**
- **Maintainability**
  - Explicit dependences

© 2019 IBM Corporation

# Example of Challenge

**The Diary of a Datum:**

An Approach to Modeling Runtime Complexity in Framework-Based Applications

- Nick Mitchell, Gary Sevitsky, Harini Srinivasan
- Workshop on Library-Centric Software Design, San Diego, CA, 2005

bytes (SOAP) → Parse, set field in business object → Calendar* (Java field) → Copy to another version of the business object → Date* (Java field) →

Cost:
- 268 calls
- 70 objects

10,000+ instructions

*new objects

**Figure 1: A dataflow diagram of how the Trade benchmark transforms a date, from a SOAP message to a Java object.**

© 2019 IBM Corporation

# Inefficiency and Intractability

*What compiler can analyze all these steps and parallelize them – and/or do dead code elimination?*

Parse (using SOAP CalendarDeserializer)

Set business object field via reflection

Cost:
- 6 calls
- 1 object

build Calendar

Cost:
- 15 calls
- 15 objects

Calendar*
+ 11 arrays*
+ TimeZone*

2 longs
(TZ and millis)

SimpleDateFormat
+ Calendar

ParsePosition*

bytes (SOAP)

extract value from SOAP tag

String*

parse time zone and millis; reformat without them

String*

TimeZone*
(constant)

parse using Simple-Date-Format

Date*

add in timezone and millis

Date

set time

Calendar

box into array

Object[]*

call invoke() on setter

Calendar

Cost:
- 30 calls
- 3 objects

Cost:
- 11 calls
- 6 objects

Cost:
- 95 calls
- 39 objects

Cost:
- 4 calls
- 0 objects

Cost:
- 7 calls
- 1 object

get schema info

XML and Java types

Cost:
- 10 calls
- 0 objects

get de-serializer

Deserializer*

Cost:
- 51 calls
- 5 objects

*new objects

BeanPropertyDescriptor

**Figure 2. Zooming in on the first step of Figure 1 shows how the SOAP framework transforms the purchase date field.**



© 2019 IBM Corporation

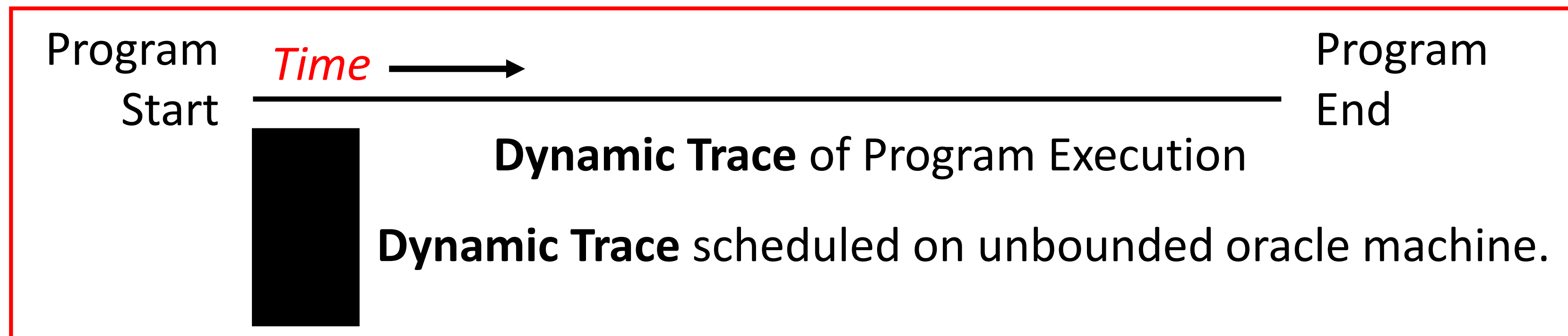
# Some Additional Dataflow Challenges

- **Explicitly pass all state (cumbersome)**
- **Programmer mental models**
- **Handling overabundance of parallelism at peak**
- **Interoperation with other code**
- **Interaction with non-dataflow I/O**
- **Applicability to random code**
- **Memory Footprint**
- **Wide separation of parallelism in code**
  - Not tractable for compilers or processors
- **Compiler analysis typically ignorant of semantic function**

© 2019 IBM Corporation

# High "Oracle Parallelism"

- Many studies have found large amounts of parallelism, even in integer apps like SPECint:
  - 1970, Tjaden and Flynn
  - 1972, Riseman and Foster
  - 1981, Nicolau and Fisher
  - 1991, Wall
  - 1991, Butler et al
  - 1992, Austin and Sohi
  - 1992, Lam and Wilson
  - 1992, Theobald et al
  - 1993, Rauchwerger et al
  - 1998, Postiff et al
  - 1999, Ebcioglu et al
- **Machine with unbounded resources and an oracle for branch prediction and memory disambiguation could execute *hundreds or thousands of instructions per cycle*.**



© 2019 IBM Corporation

# Natural Structure → Parallelism

- Any long running program, must have the following structure at some level:

```
while (end_cond_not_met) {
        task_1 ();
        task_2 ();
        …
        task_n ();
}
```

1. Tasks *1-n* are often largely independent of each other, or
2. Task *q* at iteration **i** is independent of task *q* at iteration **i+1**.

© 2019 IBM Corporation

# Example

*LLVM* (mostly) compiles one function at a time.

- Compilation of each function is largely independent.
- A skilled programmer could parallelize *LLVM* to compile functions in parallel.
- *Better to perform this parallelization:*
  - *Automatically or with tools providing guidance.*
  - *For any program, not just LLVM.*



© 2019 IBM Corporation

# Parallelism vs # of Studies

Minimum Parallelism of Any Benchmark (from Max Parallelism found in any Study)

900
800
700
600
500
400
300
200
100
0

1 2 3 4 5 6 7

# of Studies

When good techniques from many researchers are applied:

→ Almost all apps appear to have more than 500-way parallelism.

© 2019 IBM Corporation

# 5 < Parallelism < 50
## (Highest Parallelism in Any Study)

Degree of Parallelism

50
40
30
20
10
0

ccom
tpcc
wave5
egrep
backprop
grr
Linpack
sed
metronome
yacc
uncompress
Stanford
eco
Whetstone
Livermore

"Low" Parallelism only in limited apps .

These estimates are probably far too low.

© 2019 IBM Corporation

# 50 < Parallelism < 500
## (Highest Parallelism in Any Study)

Degree of Parallelism

500
400
300
200
100
0

go
m88ksim
irsim
awk
hydro2d
perl
tex
latex
ijpeg
nasker

High degree of parallelism in wide variety of apps.

© 2019 IBM Corporation

# 500 < Parallelism < 5000
## (Highest Parallelism in any Study)

Degree of Parallelism

5000
4000
3000
2000
1000
0

mgrid
turb3d
applu
li
vortex
apsi
ora
spice2g6
su2cor
gcc
espresso
compress
swim
mdljsp2
doduc
mdljdp2

Parallelism > 500 even for irregular integer apps:

**li, gcc, compress**

© 2019 IBM Corporation

# Parallelism > 5000
# (Highest Parallelism in any Study)



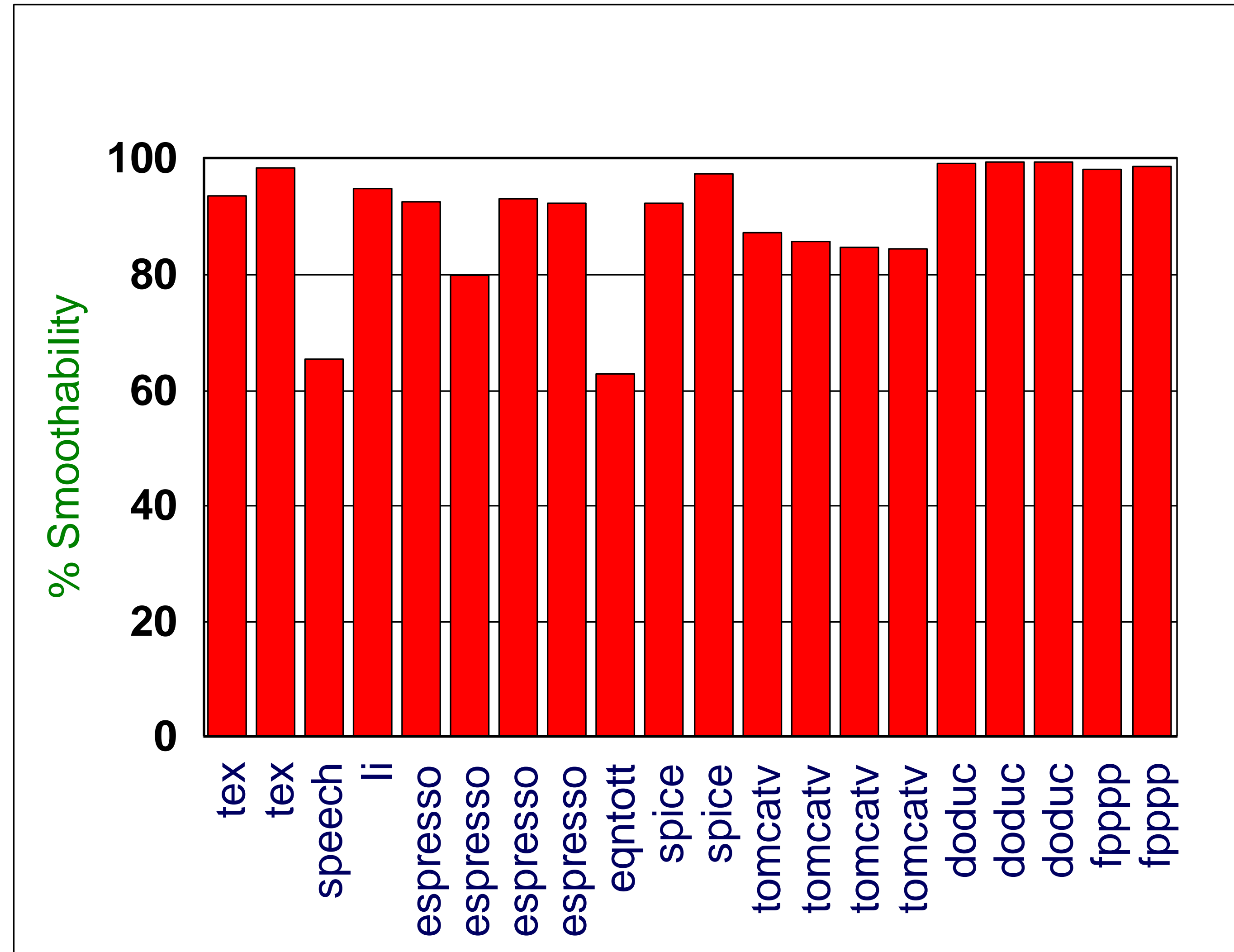
Highest Parallelism in Numerical Applications:

*5,800 - 188,000 IPC*

© 2019 IBM Corporation

# Oracle Parallelism: Smoothability

- **Does high parallelism require unbounded hardware:**
  - Can all the parallelism be effectively smoothed out onto finite hardware?
  - And still run in almost as short a time?




*Theobald, Gao, Hendren*
*Micro 1992*



© 2019 IBM Corporation

# Some Efforts to Exploit Coarser-Grain Parallelism

- **Wisconsin:** **MultiScalar**
- **UIUC, CMU, Stanford:** **Speculative Multithreading**
- **UPC:** **Kilo Processors**
- **Cornell:** **Cherry**
- **Princeton:** **DSWP, Commutativity**

- ***<Many>:*** ***Transactional Memory***

*+ Many Others*

© 2019 IBM Corporation

# Exploiting Coarse-Grained Single-Thread Parallelism

**Q:** Why have we not already exploited this task level parallelism in a single thread?

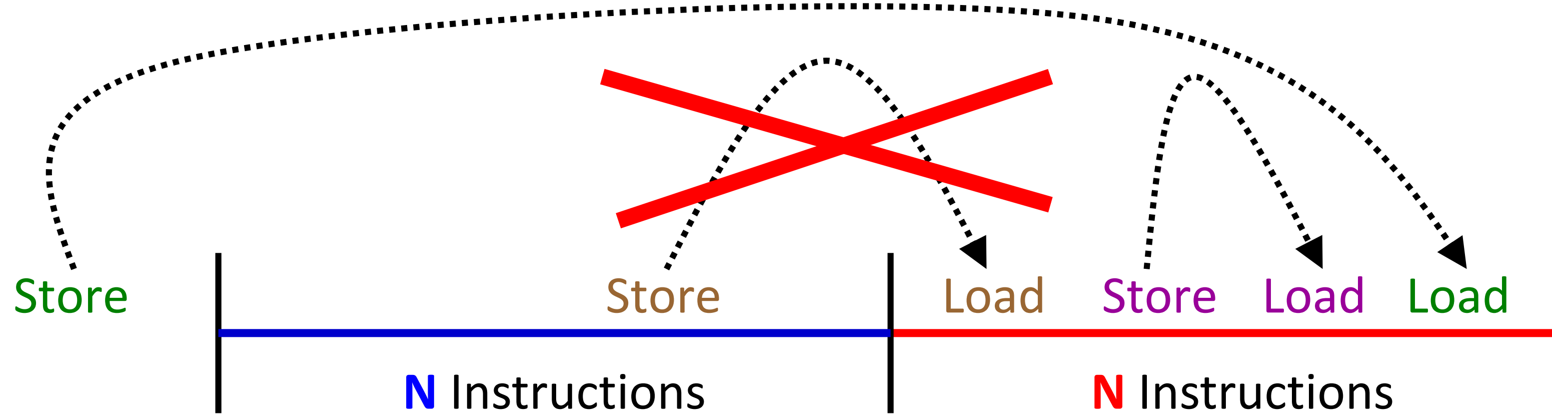
**A:** It is widely separated – independent compute often millions of instructions apart.

- **Too hard** for a compiler:
  - To generate threads from arbitrary code.
  - To determine all memory aliases.
  - To know full call graph:
    - Indirect method calls
    - Dynamically linked libraries
- **Too big** for the largest *instruction window / reorder buffer*.
- **Too hard** to predict branches:
  - **90%** chance of reaching of following correct path for a million ins
    ➔ **99.9999+%** individual branch prediction accuracy.



© 2019 IBM Corporation

# Memory Parallel Regions

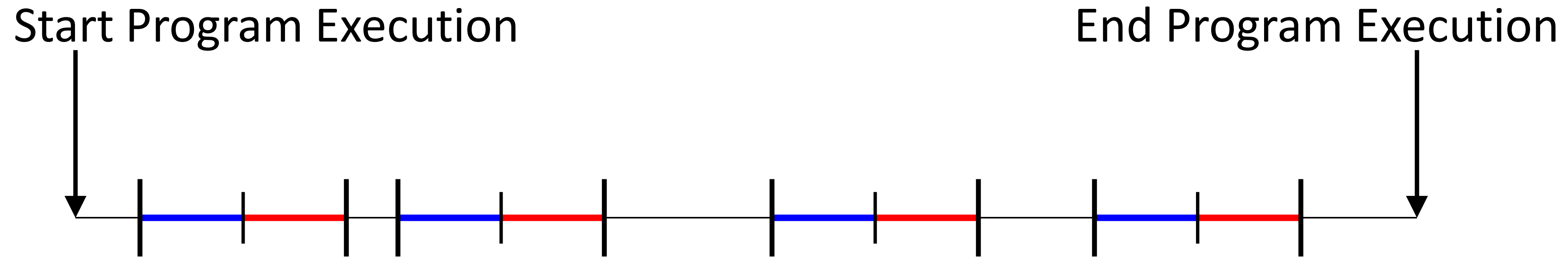



Find regions of **N** instructions that are independent of stores performed in the previous **N**-instruction region.

➔ Execute Blue and Red Regions in Parallel

© 2019 IBM Corporation

# Covering Execution with Memory Parallel Regions




For a given **N**, how much of execution is covered by independent region pairs?

© 2019 IBM Corporation

# Livermore Loops

% of Execution Covered by N-instruction Regions Independent of Preceding N Instructions

100
80
60
40
20
0

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

4K Ins
64K Ins

© 2019 IBM Corporation

# Dependences in 4K vs 64K

Start Program Execution

End Program Execution

Store Load

Store Load

4K 4K

64K

64K



© 2019 IBM Corporation

# go

% of Execution Covered by N-instruction Regions Independent of Preceding N Instructions

100
80
60
40
20
0

4K
64K
2M

© 2019 IBM Corporation

## Stanford Integer Benchmarks

% of Execution Covered by N-instruction Regions Independent of Preceding N Instructions

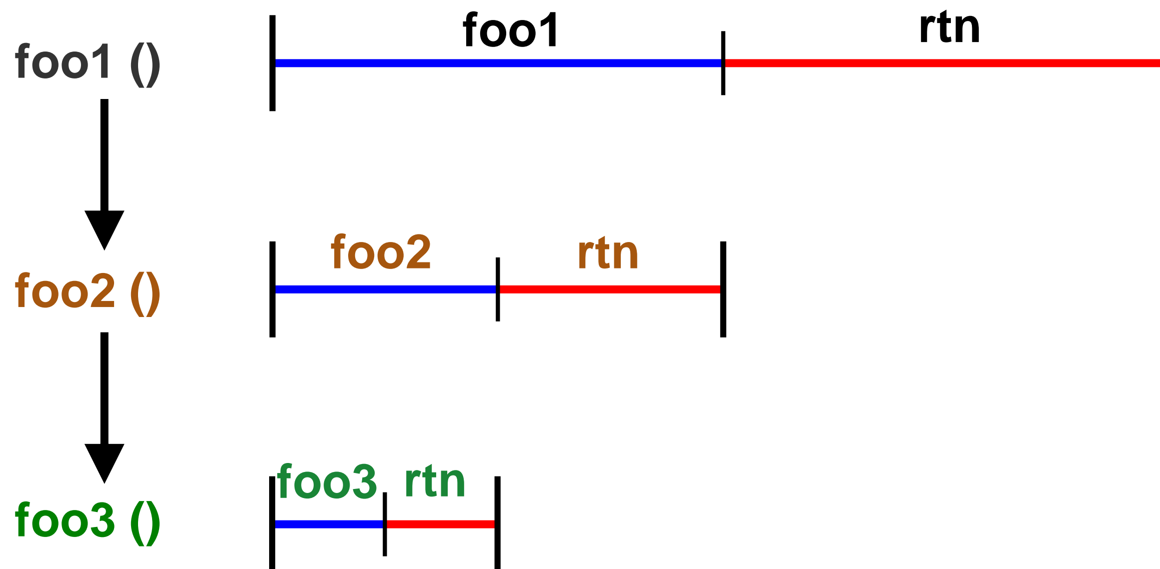
100
80
60
40
20
0

4K
64K
2M

© 2019 IBM Corporation

# Exploiting Memory Parallel Regions

1. **Find** long running functions.
2. **Determine** subset of those functions where code at the return point is independent of the results produced by the function.




*Combination of hardware capabilities and software tools may be most effective.*

© 2019 IBM Corporation

# Some Additional Dataflow Challenges

- **Explicitly pass all state (cumbersome)**
- **Programmer mental models**
- **Handling overabundance of parallelism at peak**
- **Interoperation with other code**
- **Interaction with non-dataflow I/O**
- **Applicability to random code**
- **Memory Footprint**
- **Wide separation of parallelism in code**
  - Not tractable for compilers or processors
- **Compiler analysis typically ignorant of semantic function**

© 2019 IBM Corporation

# DSLs – Domain Specific Languages

**Example – PLDI'2018:**
*Spatial:*
*A Language and Compiler for Application Accelerators*

- D Koeplinger, M Feldman, R Prabhakar, Y Zhang, S Hadjis, R Fiszel, T Zhao, L Nardi, A Pedram, C Kozyrakis, K Olukotun

Good, but compiler still does not know that this code implements **merge_sort**

- Global optimizations easy to miss
- No opportunity to substitute another type of sort

```
1  def Merge_Sort(offchip: DRAM[Int], offset: Int) {
2    val N = 1024  // Static size of chunk to sort
3    Accel {
4      val data  = SRAM[Int](N)
5      data load offchip(offset::N+offset)
6
7      FSM(1){m => m < N}{ m =>
8        Foreach(0 until N by 2*m){ i =>
9          val lower = FIFO[Int](N/2).reset()
10         val upper = FIFO[Int](N/2).reset()
11         val from  = i
12         val end   = min(i + 2*m - 1, N) + 1
13
14         // Split data into lower and upper FIFOs
15         Foreach(from until i + m){ x =>
16           lower.enq(data(x))
17         }
18         Foreach(i + m until end){ y =>
19           upper.enq(data(y))
20         }
21
22         // Merge of the two FIFOs back into data
23         Foreach(from until end){ k =>
24           val low  = lower.peek() // Garbage if empty
25           val high = upper.peek() // Garbage if empty
26           data(k) = {
27             if      (lower.empty) { upper.deq() }
28             else if (upper.empty) { lower.deq() }
29             else if (low < high)  { lower.deq() }
30             else                  { upper.deq() }
31           }
32         }
33       }
34     }{ m => 2*m /* Next state logic */ }
35
36     offchip(offset::offset+N) store data
37   }
```

# Higher-Level Data Types and Algorithms as First-Class Objects

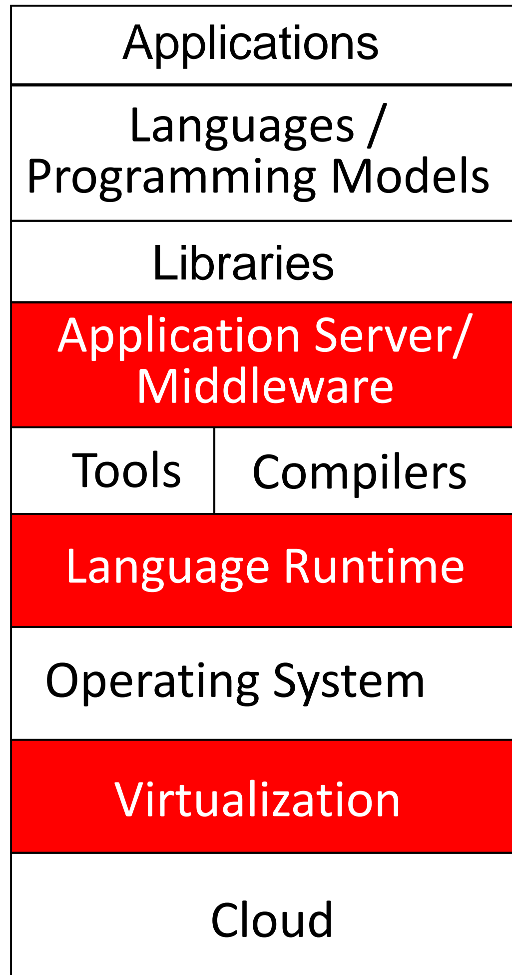
- Example: Repeated Insertion to a List, with sort immediately after each insertion
  - List-insert / Sort
  - List-insert / Sort
  - List-insert / Sort
- But knowing the semantics of List, Insert, and Sort allows optimization
  - List-insert
  - List-insert
  - List-insert
  - Sort
- Many higher-level constructs could be represented to compiler:
  - Lists, trees, graphs, heaps, stacks, etc
  - Dates and times
  - Images
  - …

© 2019 IBM Corporation

# But What if Rewriting Large Amounts of Code is not an Option?

© 2019 IBM Corporation

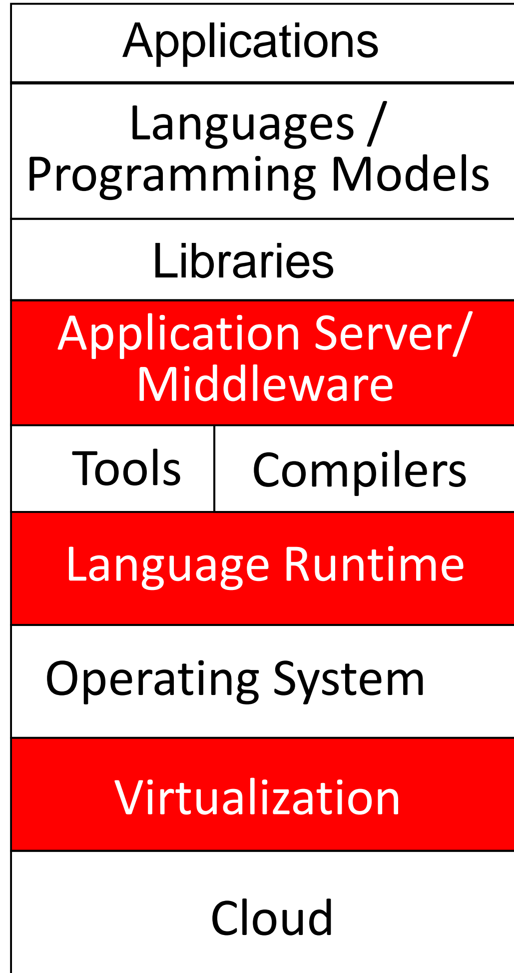
# The Stack: Benefit, Bane, and Exploit




- To address complexity we created runtime stack
  - *Layer below provides abstraction to layer above*
- Widely successful
  - Hides most details of each layer
  - Enables componentization
    - Ability to change components at layer → Incentive for improvement
- Limitations
  - More layers accrue over time (e.g. *JVM, App server, hypervisor*)
    - 1 level of indirection → Brilliance
    - N levels → ???
  - Thin interfaces between layers promote
    - Lack of synergy
    - Duplicate functionality
      - *AppServer, JVM, OS, Virtualization, HW* have thread abstraction

© 2019 IBM Corporation

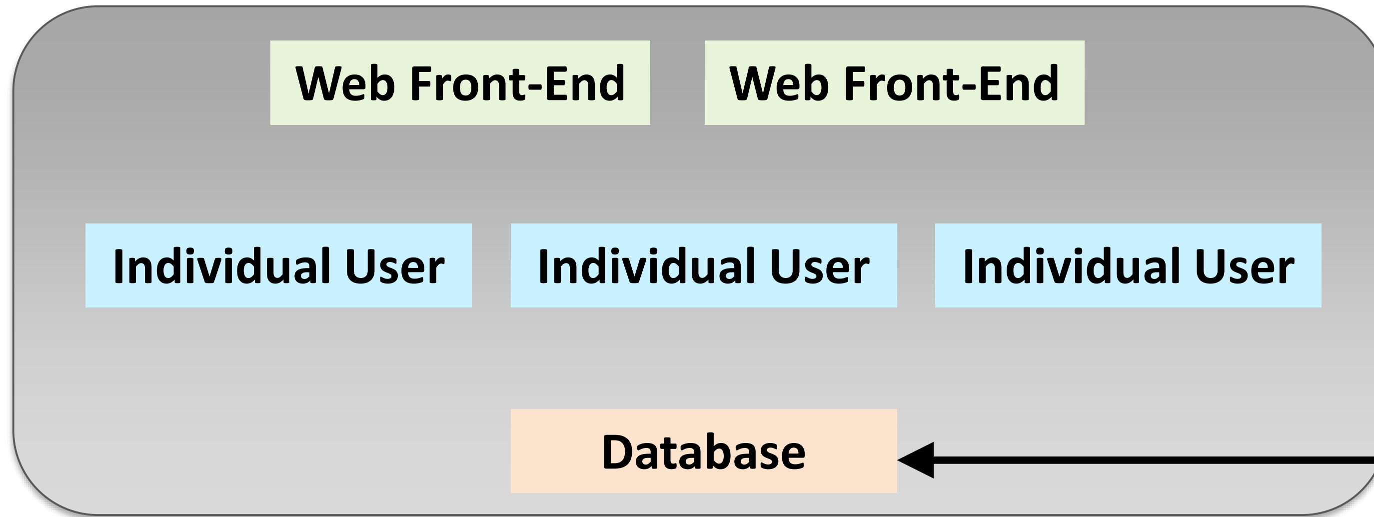
# The Stack: Where to Focus?




← **Here**

© 2019 IBM Corporation

# Cloud == Dataflow?




*Only point of sequential code – isolated and minimized*

**CASE 1:** Single Enterprise in Cloud handling many individual users.

- Separate, Isolated VMs
- Processes / Containers

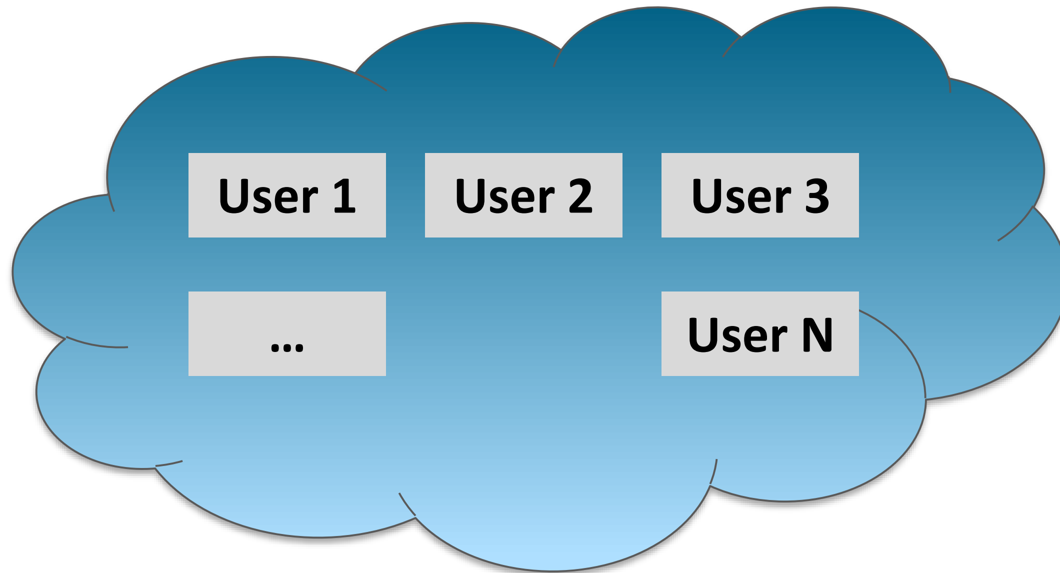
**Result – Dataflow:**

- Scaleout Model with seamless scale-up and scale-down as degree of parallelism varies
- Arbitrary number of independent program counters
- Varying use of processors as parallelism varies
- Isolated Data

© 2019 IBM Corporation

# Cloud == Dataflow?

**CASE 2:** Broad Set of Enterprises in Cloud




Each user can be dataflow within themselves
- *As in previous slide*

**User** = Broad set of services and tasks controlled by one entity

**Result – Dataflow:**
- Lots of program counters
- Lots of processors
- No global state across users
- Run as data / input becomes available

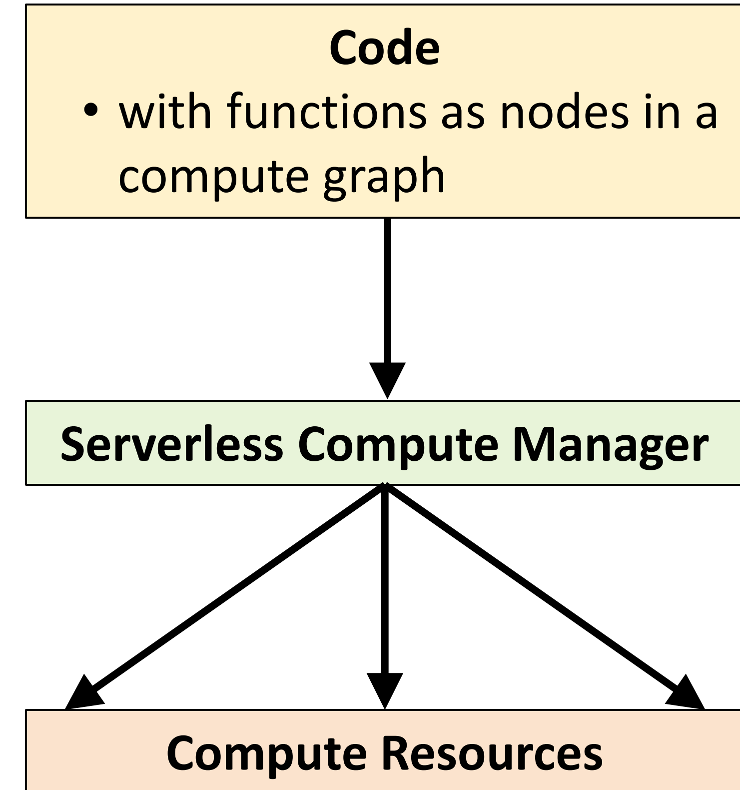
© 2019 IBM Corporation

# Cloud == Dataflow?



## CASE 3: Serverless Computing as Dataflow

- Amazon Lambda
- Google Cloud Platform – Functions
- Microsoft Azure
- IBM Cloud Functions
- Openwhisk
- Kubeless
- …




**Result – Dataflow:**

- Run as data / input becomes available
- Lots of program counters
- Lots of processors
- No global state across users

© 2019 IBM Corporation

*Is cloud an unappreciated commercial success for dataflow and extreme scale?*

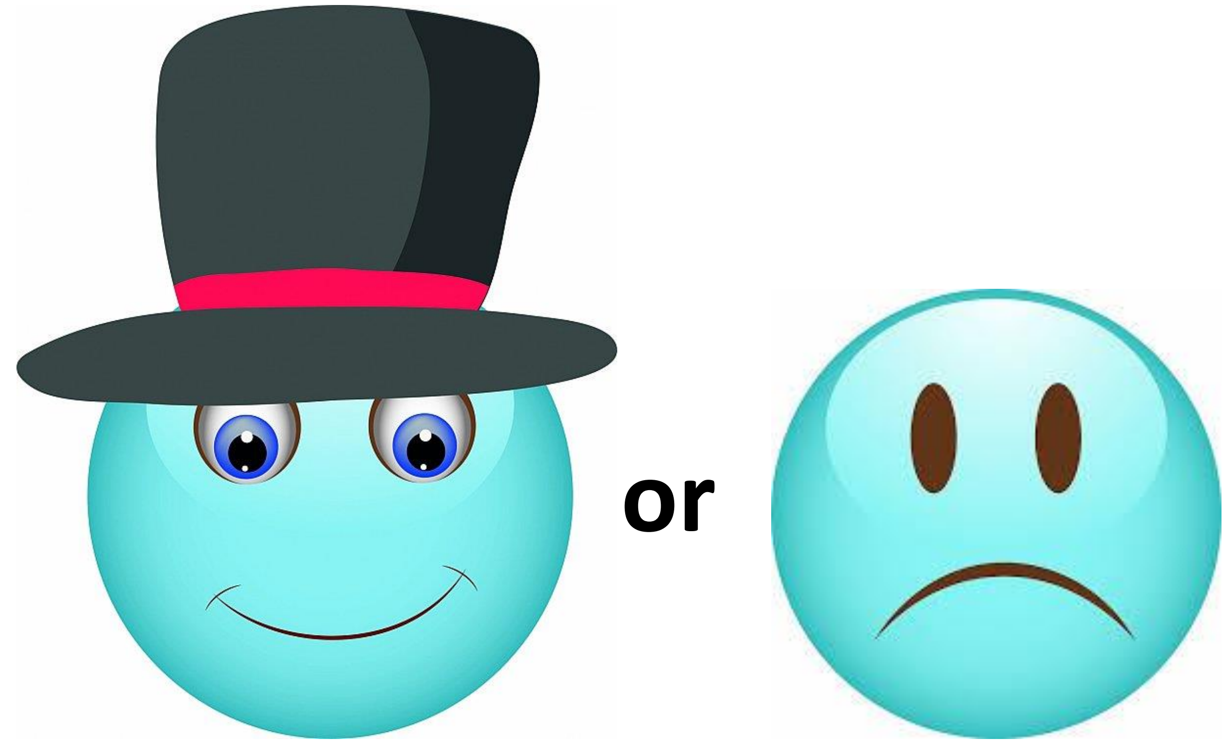
© 2019 IBM Corporation

# Lessons: Dataflow and Cloud

- **There has long been debate about the proper granularity to express dataflow ops**
- **Cloud shows one case where coarse-grain allows a massive dataflow approach to succeed commercially and attract a vast set of users**
  - As programmers
  - As users of the end-result
- **What lessons from cloud can be applied elsewhere in dataflow?**
  - Need easy-to-use frameworks
  - Need massive independent parallel threads on which to run
  - Need massive hardware processing resources
  - Need coordination by centralized mechanisms to be too complex to implement

© 2019 IBM Corporation

# Dataflow and Security




- **Good**
  - No pointers
  - No buffer over-run issues
- **Bad**
  - Side channels like Spectre and Meltdown seem open
- **Indifferent**
  - Phishing and Spear-phishing
  - Dumpster diving

© 2019 IBM Corporation

# Conclusions

- **Worst of Times**
  - No commercial takeoff of traditional dataflow despite unprecedented compute resources, new computing paradigms, more programmers than ever, more languages than ever
- **Best of Times**
  - Major tenets of dataflow have been embraced commercially in cloud at extreme scale
  - They are succeeding
- **Next Steps**
  - What do we learn from those successes?
  - How do we amplify them?
  - What other fields are ripe?

© 2019 IBM Corporation