This Architecture Tastes Like Microarchitecture

Curtis Dunham

Arm Research

Pioneering Processor Paradigms (WP3)

Sunday, February 25, 2018

Disclaimer

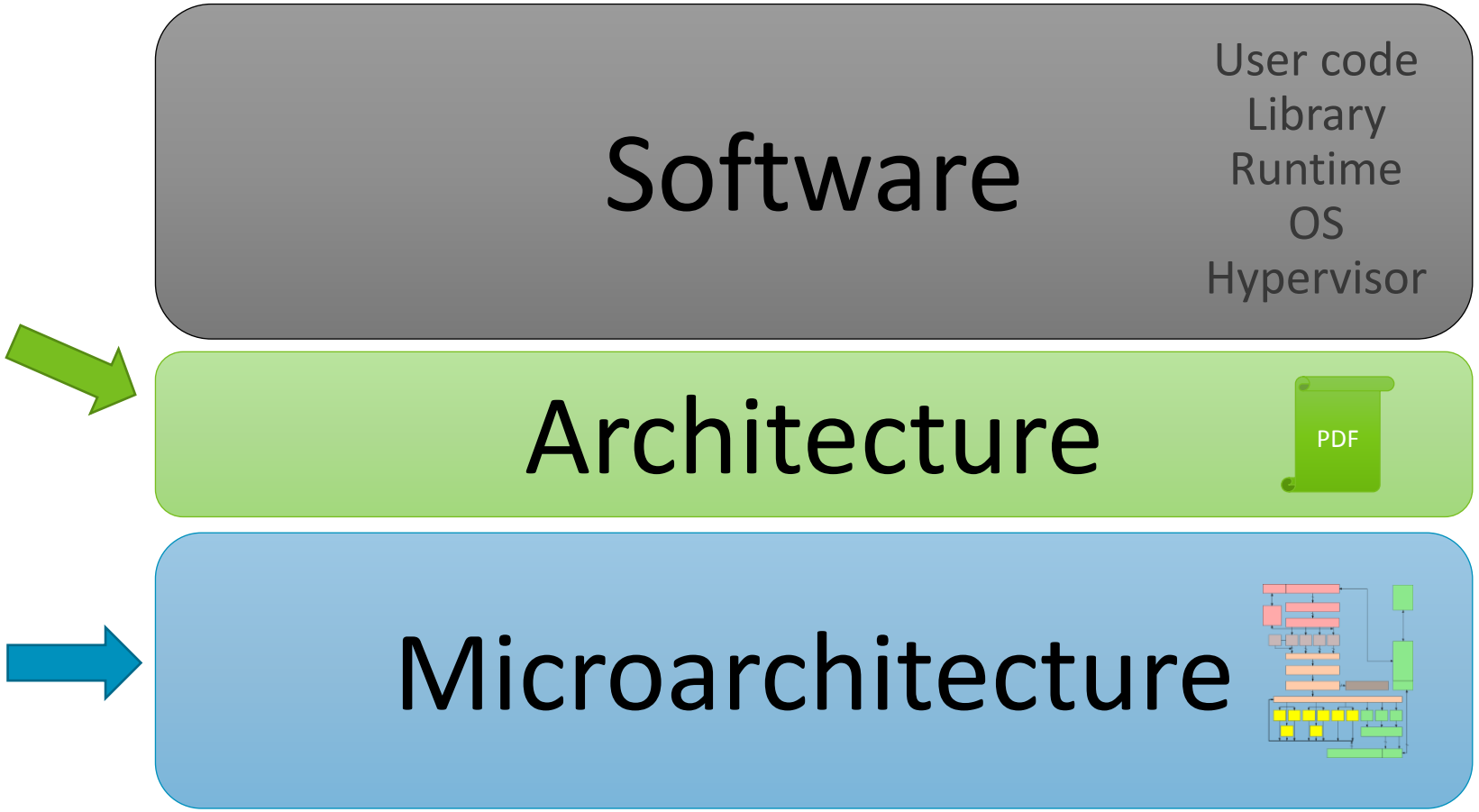
I do not speak for my employer.

This presentation (and accompanying paper) do not represent Arm projects or future products.

Clarification: definitions of terms

Architecture:
Abstraction provided to software;
the hardware-software interface;
the instruction-set architecture (ISA).

Microarchitecture:
Implementation of the architecture abstraction.



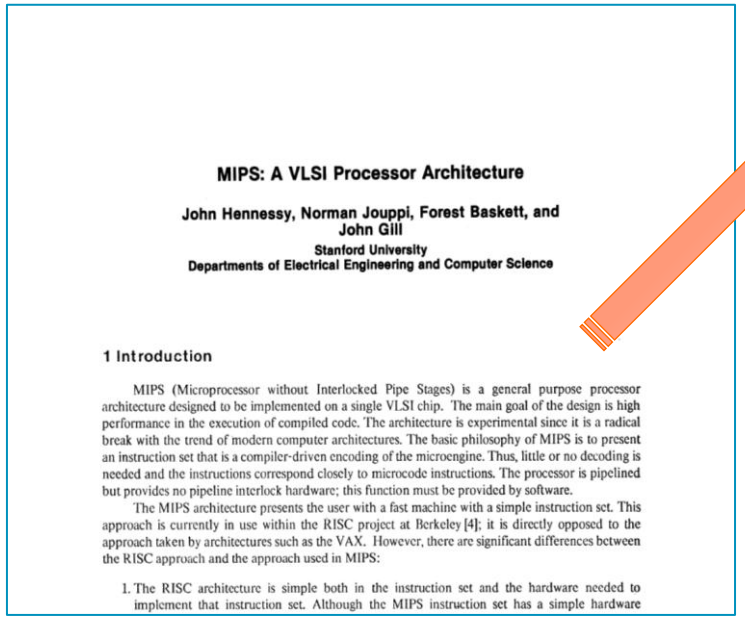
arm

The paper

Future ISA Design

Time multiplexing

Operand encoding



MIPS: A VLSI Processor Architecture

John Hennessy, Norman Jouppi, Forest Baskett, and John Gill

Stanford University

Departments of Electrical Engineering and Computer Science

1 Introduction

MIPS (Microprocessor without Interlocked Pipe Stages) is a general purpose processor architecture designed to be implemented on a single VLSI chip. The main goal of the design is high performance in the execution of compiled code. The architecture is experimental since it is a radical break with the trend of modern computer architectures. The basic philosophy of MIPS is to present an instruction set that is a compiler-driven encoding of the microengine. Thus, little or no decoding is needed and the instructions correspond closely to microcode instructions. The processor is pipelined but provides no pipeline interlock hardware; this function must be provided by software.

The MIPS architecture presents the user with a fast machine with a simple instruction set. This approach is currently in use within the RISC project at Berkeley [4]; it is directly opposed to the approach taken by architectures such as the VAX. However, there are significant differences between the RISC approach and the approach used in MIPS:

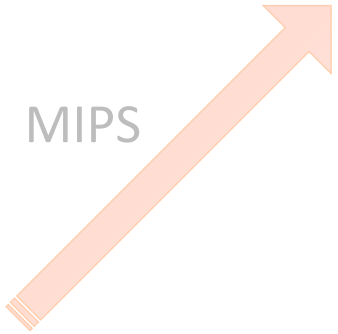
1. The RISC architecture is simple both in the instruction set and the hardware needed to implement that instruction set. Although the MIPS instruction set has a simple hardware

This talk

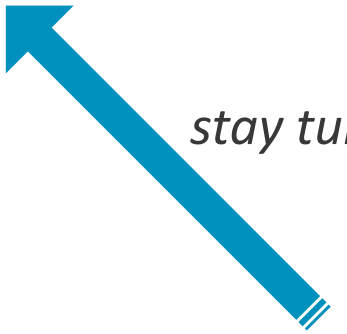
Future ISA Design

Time multiplexing

Operand encoding



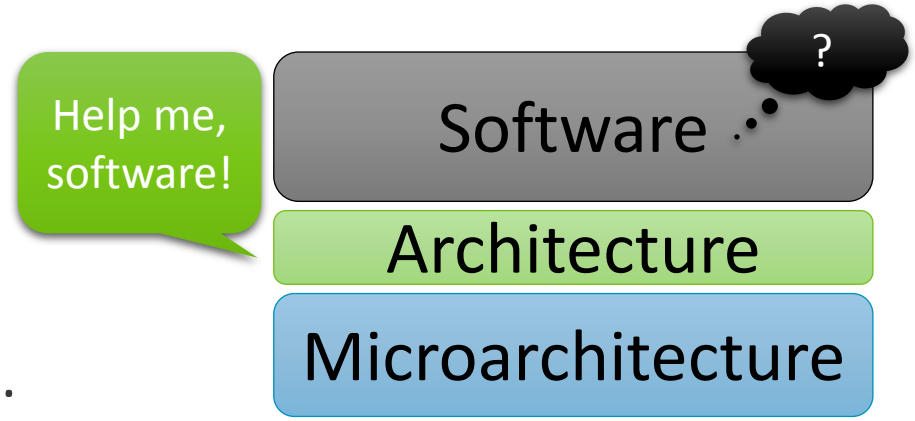
MIPS



stay tuned!

A different route to the same destination

On the resourcefulness of Computer Architects



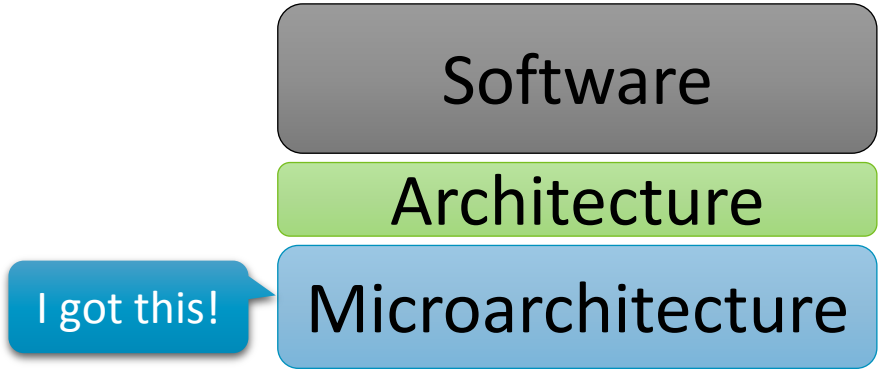
We have a history of trying to solve microarchitectural difficulties by exposing them to software.

Obvious example: Branches are really hard, so how about…

- Delay slots
- Branch hints

Then! We solve the problem better in a transparent way.

- Front end microarchitecture, *e.g.* branch prediction



… Now the ISA contains antiquated performance hacks.

And maybe worse things…

arm

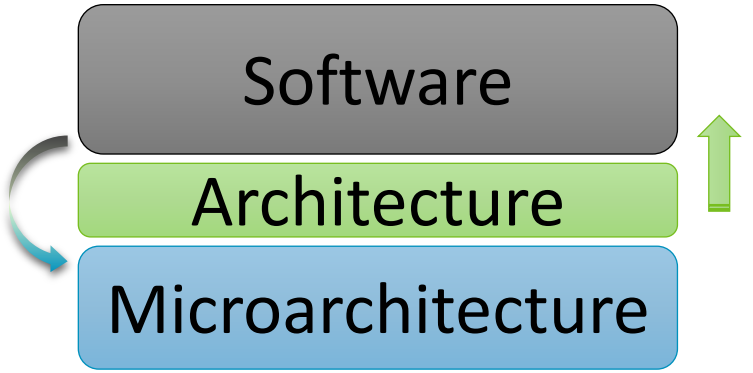
On the future resourcefulness of Computer Architects

Other ISA aspects follow the same pattern, but they aren't as obvious.

They seem normal.

These unchallenged assumptions are slowing the rate of progress; architecture *must* continue:

- gaining ground against the software stack
- raising the level of abstraction
- re-negotiating the hardware-software boundary.

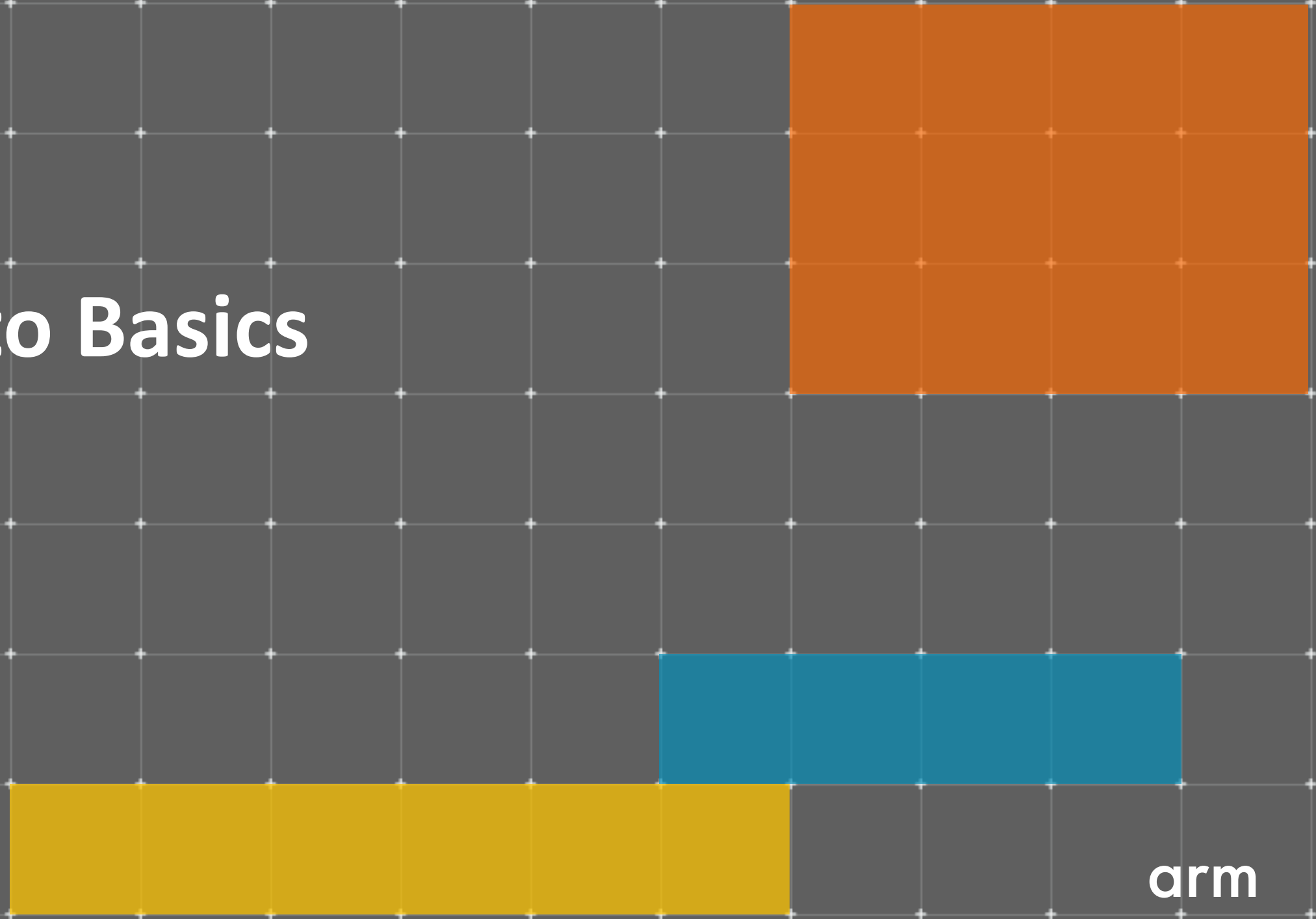


This talk: some promising avenues on important problems!

But first, a little history…

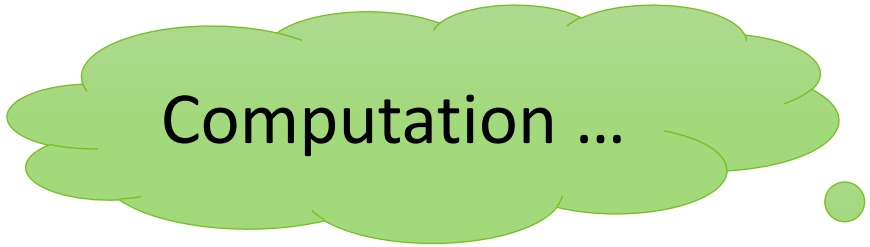
arm

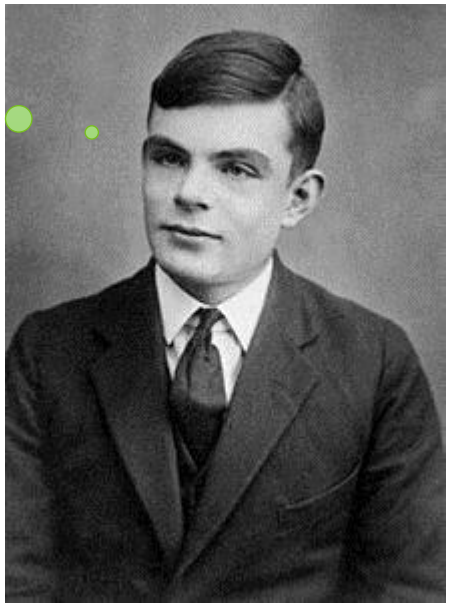
Back to Basics



arm

Turing machine



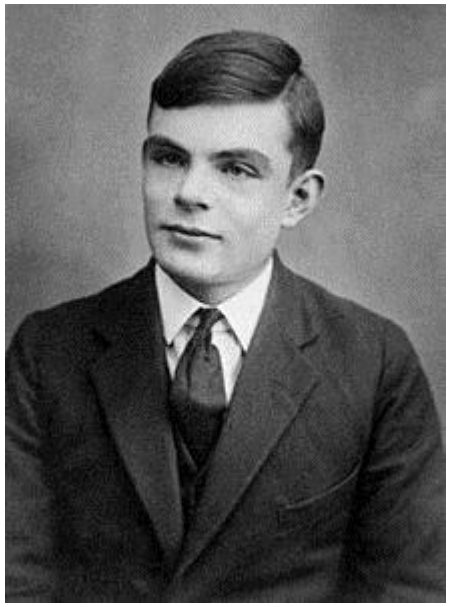


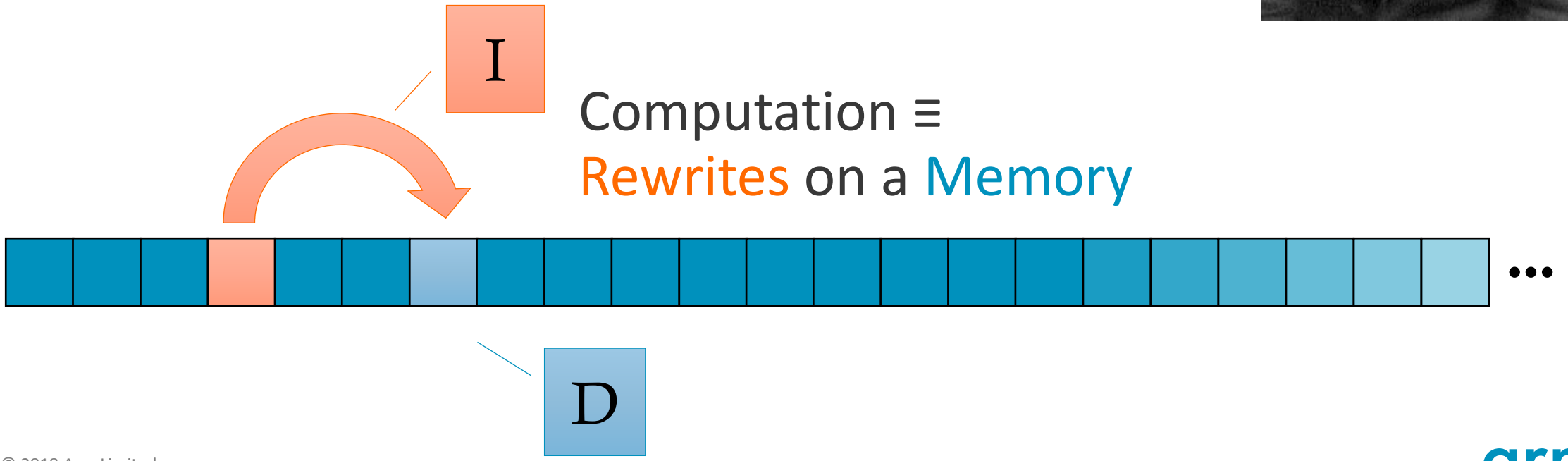
Alan Turing

Universal Turing Machine

arm

Turing machine retrospective



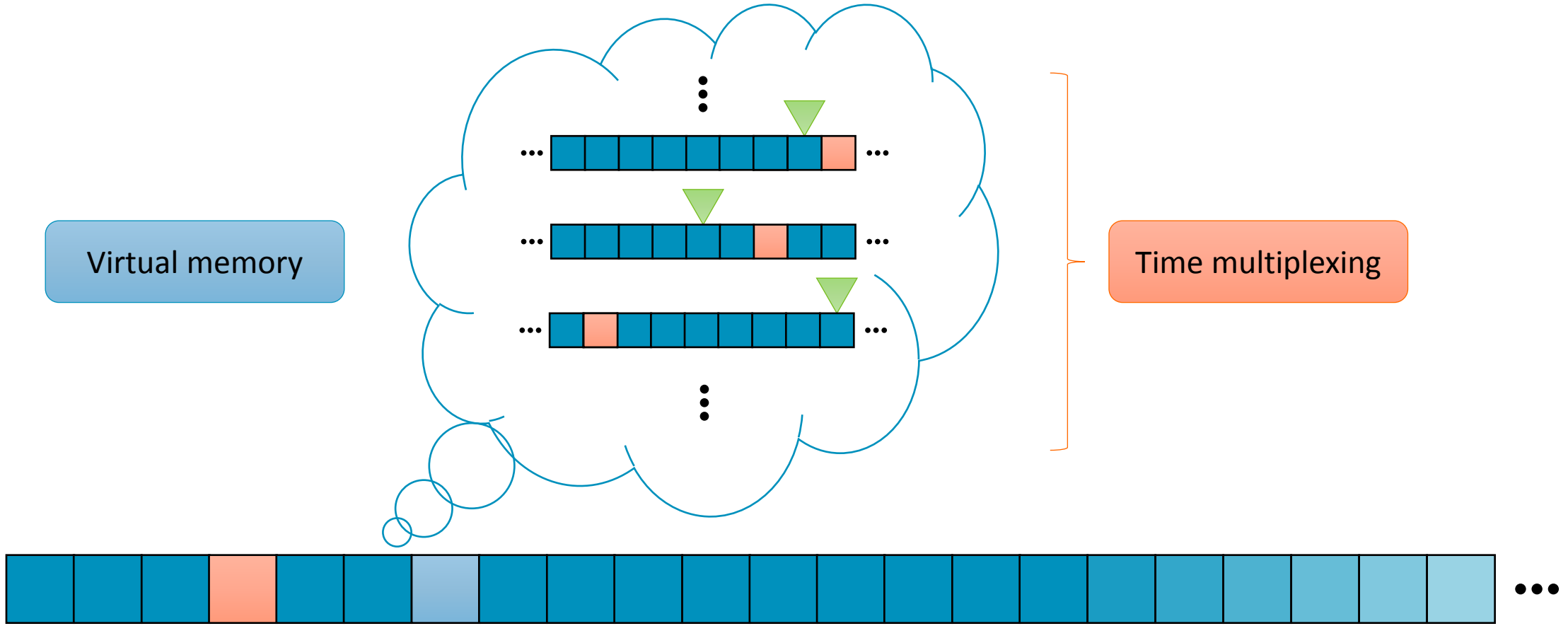


Computation ≡
Rewrites on a Memory

arm

Virtual computation: within one computer, many

Nota bene: not “virtualization” per se; think “processes”



arm

Virtual computation basics

Data vs. Code, in search of programming model goodness

Virtual memory

Use of data is unevenly distributed

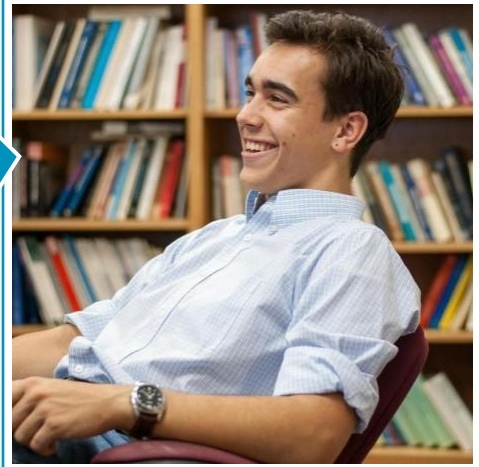
- Want: system automatically maps data to devices (*i.e.*, optimizes locality)

Virtual computation

Use of code is unevenly distributed (code is data…)

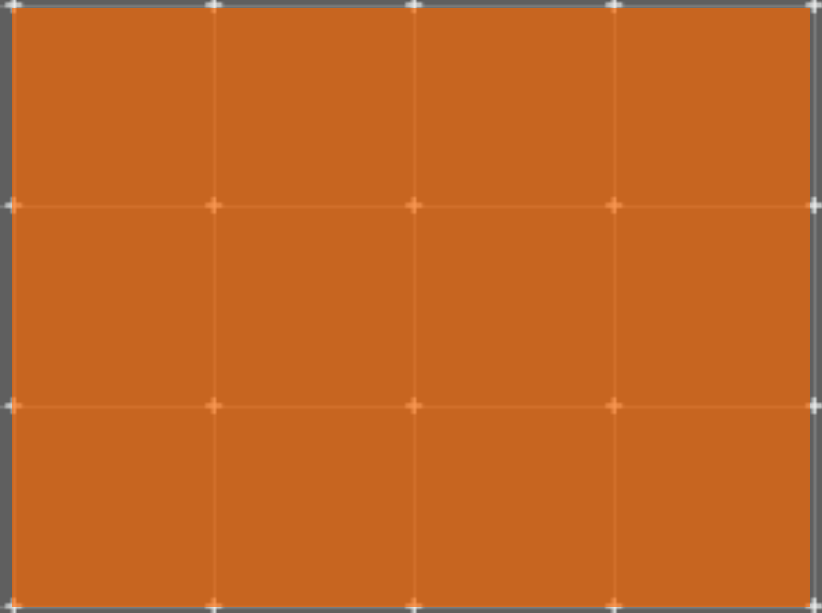
- Want: system automatically maps code to devices (*i.e.,* optimizes responsiveness and/or throughput)

"Computer, do for me!"



arm

A story of virtual memory



arm

Virtual memory, a.k.a. “folding”

Is automatic “folding” of programs efficient enough to displace manual?, Sayre (IBM), 1969

Abstract:

The operation of “folding” a program into the available memory is discussed. Measurements by Brawn et al. and by Nelson on an automatic folding mechanism of simple design, **a demand paging unit** built at the IBM Research Center by Belady, Nelson, O'Neill, and others, permitting its quality to be compared with that of manual folding, are discussed, and it is shown that given some care in use the unit performs satisfactorily under the conditions tested, even though it is operating across a memory-to-storage interface with a very large speed difference. The disadvantages of prefolding, which is required when the folding is manual, are examined, and a number of the important troubles which beset computing today are shown to arise from, or be aggravated by, this source. It is concluded that a folding mechanism **will probably become a normal part of most computing systems**.

(Emphases added)

“Self-created obstacles”

“In ceasing to expend energy:

- in a process whose main result is to make programs less fit to run on other machine configurations
- to run in company with other programs, or
- to run with temporarily reduced resources,

we do more than reduce costs; **we remove self-created obstacles** which today are impeding the development of needed types of systems”

Move In

Move Out

arm

Some predictions

arm

Predictions / Recommendations 22 years hence

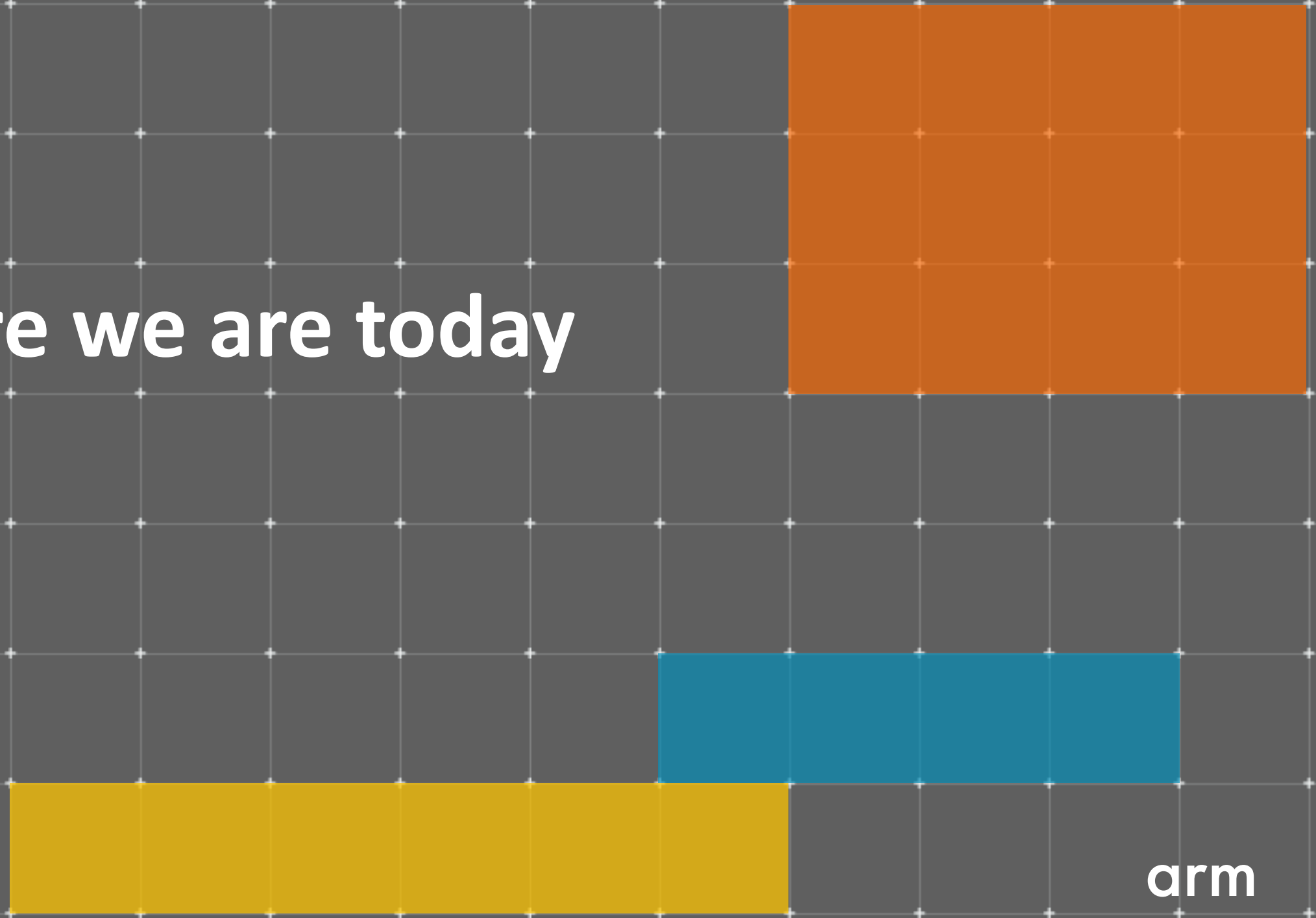
from Microprocessor Report, vol. 10 #10 (1996):

For this special issue, we asked several processor architects how, based on 25 years of history, they see the microprocessor continuing to evolve in the future. Their responses discuss several technical barriers to success and how they might be overcome. Equally important is an often overlooked issue: what will people do with all this performance?

"…over the coming decade memory subsystem design will be the only important design issue for microprocessors" – Dick Sites, "It's the Memory, Stupid!"

"Translating a clean ABI to an architecture designed as a good target is a much easier task that should approach the efficiency of code directly compiled for the hardware instruction set." – Bill Dally, "The End of Instruction Sets"

Where we are today



arm

Virtualizing techniques and mechanisms

Data vs. Code

Virtual memory: caching, paging

HW accelerated:

- Multi-level caches, cache coherence
- Translation Lookaside Buffers (TLB)
- Page table walkers (HW TLB miss servicing / fill)

Not HW accelerated:

- Paging (*i.e.* management of DRAM contents)

Issue: translation scheme (therefore their optimized embodiments) tightly coupled with primary computation modality (ISA)

Virtual computation: muxing, scheduling

HW accelerated:

- Virtual machine entry/exit, interrupt routing ("virtualization" only, limited applicability, limited deployments)

Not HW accelerated:

- Thread switching
- Thread migration
- Thread scheduling

Issue: multiplexing schemes tied to 1 computational modality (ISA, OS)

arm

Killer Microseconds

Communications of the ACM

Vol. 60 No. 4
(April 2017)

contributed articles

DOI:10.1145/3015146

Microsecond-scale I/O means tension between performance and productivity that will need new latency-mitigating ideas, including in hardware.

BY LUIZ BARROSO, MIKE MARTY, DAVID PATTERSON, AND PARTHASARATHY RANGANATHAN

Attack of the Killer Microseconds

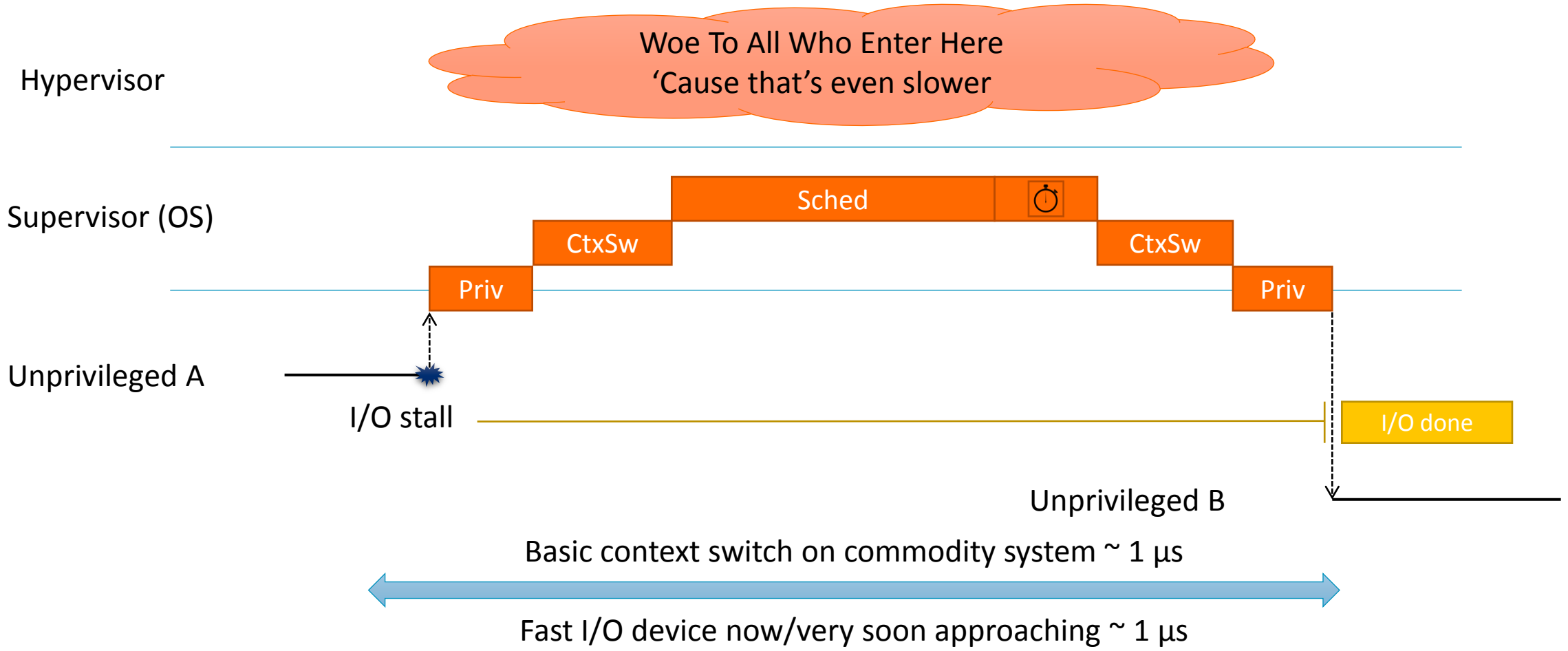
THE COMPUTER SYSTEMS we use today make it easy for programmers to mitigate event latencies in the

block a thread's execution, with the program appearing to resume after the load completes. A host of complex microarchitectural techniques make high performance possible while supporting this intuitive programming model. Techniques include prefetching, out-of-order execution, and branch prediction. Since nanosecond-scale devices are so fast, low-level interactions are performed primarily by hardware.

At the other end of the latency-mitigating spectrum, computer scientists have worked on a number of techniques—typically software based—to deal with the millisecond time scale. Operating system context switching is a notable example. For instance, when a `read()` system call to a disk is made, the operating system kicks off the low-level I/O operation but also performs a software context switch to a different thread to make use of the processor during the disk operation. The original thread resumes execution sometime after the I/O completes. The long overhead of making a disk access (milliseconds) easily outweighs the cost of two context switches (microseconds). Millisecond-scale devices are slow enough that the cost of these software-based mechanisms can be amortized (see Table 1).

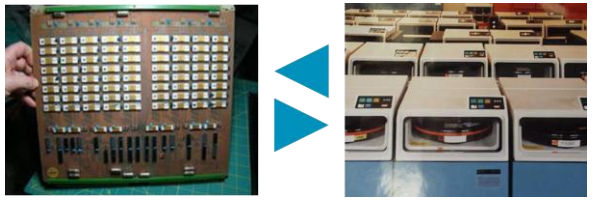
arm

Killer Microseconds cartoon

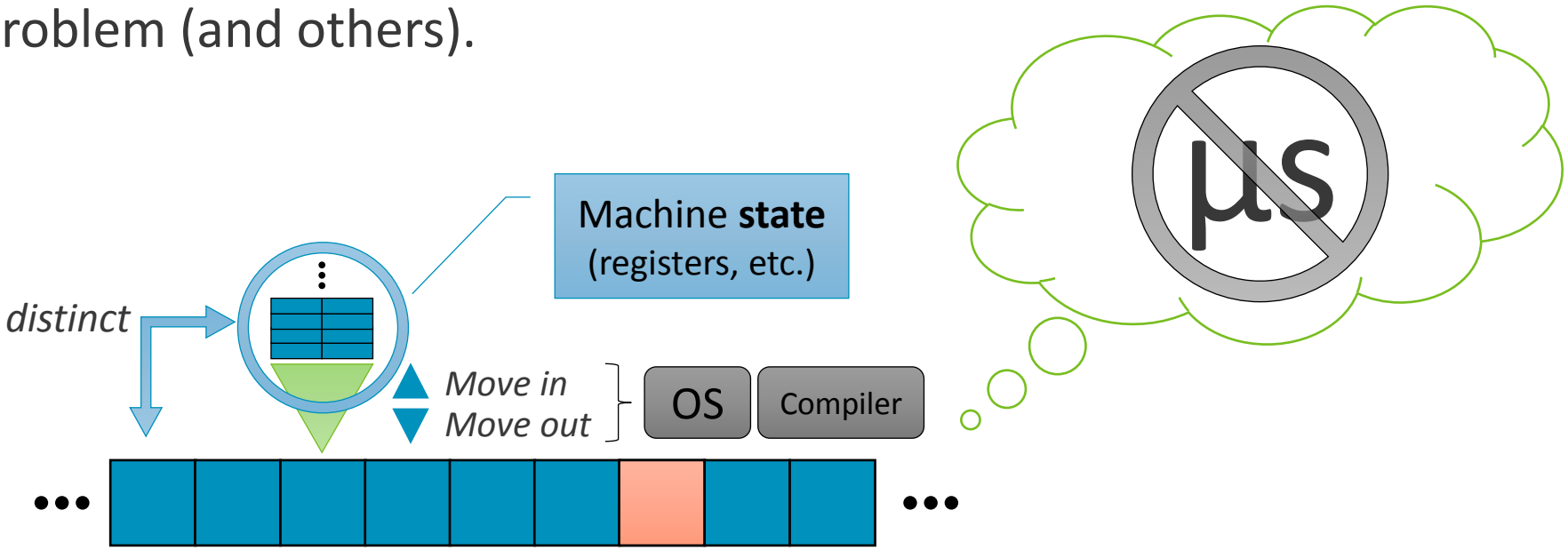


arm

The Controversial Leap: déjà vu?

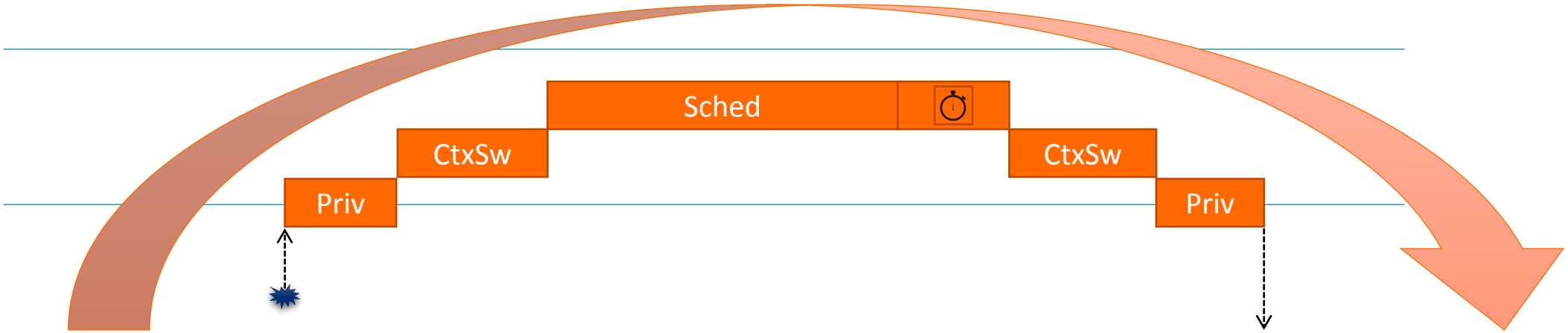


Our **architecture** “tasting like” **microarchitecture** has led to challenges solving this problem (and others).



arm

Killer Microseconds cartoon, revisited



This particular division of labor between hardware and software:

- Imposes sequence, creating a temporal bottleneck
- Needlessly involves software in a machine-specific activity (remember we dream of heterogeneous…)
- Prevents hardware tricks: asynchrony, laziness, …

arm

Registers

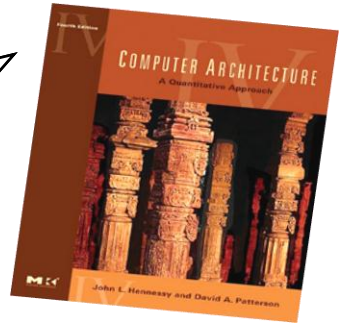
Architecture "register"

vs.

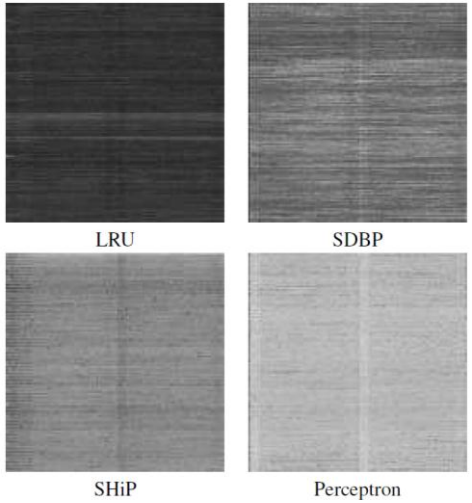
"register" Microarchitecture

Graph coloring RA	Linear Scan RA
Register renaming	ARF or PRF
Bypass network	Reorder Buffer

First, registers … are faster than memory.
Second, registers are more efficient ***for a compiler to use*** *than other forms of internal storage.*



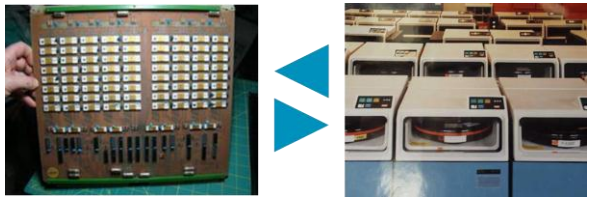
Could modern caching schemes better predict the useful lifetimes of RF/L0 D$ data than a compiler?

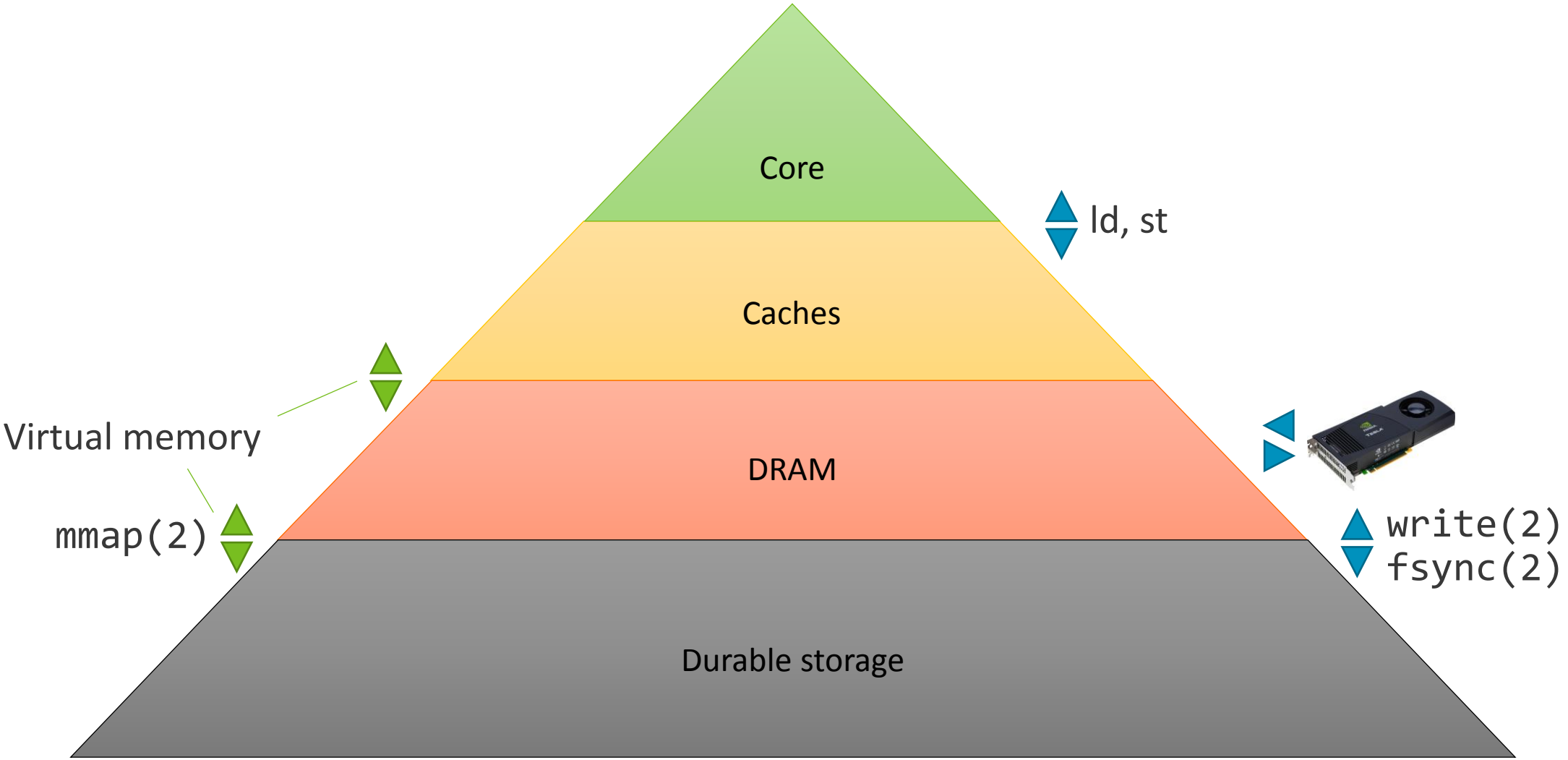


Perceptron Learning for Reuse Prediction, Teran, Wang, and Jiménez, MICRO 2016

arm

Registers as non-virtual memory





arm

“Virtual Memory” by Peter J. Denning, 1970

(abstract begins…)

The need for ***automatic storage allocation*** *arises from desires for program modularity, machine independence, and resource sharing. Virtual memory is an elegant way of achieving these objectives. In a virtual memory, the* addresses a program may use to identify information are distinguished from the addresses the memory system uses to identify physical storage sites*, and program-generated addresses are* ***translated automatically*** *to the corresponding machine addresses. Two principal methods for implementing virtual memory,* ***segmentation*** *and* ***paging****, are compared and contrasted.*

Might we add: **memory renaming** as another principal method?

(as espoused in works by Franklin *et al.*, Moshovos *et al.*, Tyson *et al.*, …)

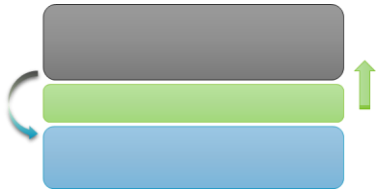
The future demands heterogeneous computing

Today: code specialized against a particular microarchitecture flavor is impractical to run on another.

CPUs and GPUs are quite different flavors.

How might we describe computations such that the code executes on the most effective computational device?

Issues include operand schemes, hardware threading, hardware scheduling, and virtual memory. The hardware-software interface must move up the stack!



Let's think about the more palatable architectures of the future.

arm

Thank You!
Danke!
Merci!
谢谢!
ありがとう!
Gracias!
Kiitos!
감사합니다
धन्यवाद

arm