Prefetching Wrap Up
Runahead as an Execution-Based Prefetcher
Runahead as an Execution-based Prefetcher

- **Idea of an Execution-Based Prefetcher:** Pre-execute a piece of the (pruned) program solely for prefetching data

- **Idea of Runahead:** Pre-execute the main program solely for prefetching data

- **Advantages and disadvantages of runahead vs. other execution-based prefetcers?**

- **Can you make runahead even better by pruning the program portion executed in runahead mode?**
  - Yes → *Continuous Runahead* is an example of this
Taking Advantage of Pure Speculation

- Runahead mode is purely speculative

- The goal is to find and generate cache misses that would otherwise stall execution later on

- How do we achieve this goal most efficiently and with the highest benefit?

- Idea: Find and execute only those instructions that will lead to cache misses (that cannot already be captured by the instruction window)

- How? → Continuous Runahead is an example of this
Continuous Runahead: Much More Efficient

- Milad Hashemi, Onur Mutlu, and Yale N. Patt,
  "Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads"
  Proceedings of the 49th International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, October 2016.
  [Slides (pptx) (pdf)] [Lightning Session Slides (pdf)] [Poster (pptx) (pdf)]
  Best paper session.

Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads

Milad Hashemi*, Onur Mutlu§, Yale N. Patt*

*The University of Texas at Austin  §ETH Zürich
Execution-based Prefetchers: Pros and Cons

+ Can prefetch pretty much any access pattern
+ Can be very low cost (e.g., runahead execution)
  + Especially if it uses the same hardware context
  + Why? The processor is equipped to execute the program anyway
+ Can be bandwidth-efficient (e.g., runahead execution)

-- Depend on branch prediction and possibly value prediction accuracy
  - Mispredicted branches dependent on missing data throw the thread off the correct execution path
-- Can be wasteful
  -- speculatively execute many instructions
  -- can occupy a separate thread context
-- Complexity in deciding when and what to pre-execute
More on Runahead Execution

- Onur Mutlu, Jared Stark, Chris Wilkerson, and Yale N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors"


One of the 15 computer arch. papers of 2003 selected as Top Picks by IEEE Micro. HPCA Test of Time Award (awarded in 2021).

Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors

Onur Mutlu §  Jared Stark †  Chris Wilkerson †‡  Yale N. Patt §

§ECE Department  †Microprocessor Research  ‡Desktop Platforms Group
The University of Texas at Austin  Intel Labs  Intel Corporation
{onur,patt}@ece.utexas.edu  jared.w.stark@intel.com  chris.wilkerson@intel.com
More on Efficient Runahead Execution

Onur Mutlu, Hyesoon Kim, and Yale N. Patt,
"Techniques for Efficient Processing in Runahead Execution Engines"

One of the 13 computer architecture papers of 2005 selected as Top Picks by IEEE Micro.

Techniques for Efficient Processing in Runahead Execution Engines

Onur Mutlu  Hyesoon Kim  Yale N. Patt

Department of Electrical and Computer Engineering
University of Texas at Austin
{onur,hyesoon,patt}@ece.utexas.edu
More Effective Runahead Execution

- Onur Mutlu, Hyesoon Kim, and Yale N. Patt, "Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns" Proceedings of the 38th International Symposium on Microarchitecture (MICRO), pages 233-244, Barcelona, Spain, November 2005. Slides (ppt) Slides (pdf) One of the five papers nominated for the Best Paper Award by the Program Committee.

Address-Value Delta (AVD) Prediction: Increasing the Effectiveness of Runahead Execution by Exploiting Regular Memory Allocation Patterns

Onur Mutlu  Hyesoon Kim  Yale N. Patt

Department of Electrical and Computer Engineering
University of Texas at Austin
{onur,hyesoon,patt}@ece.utexas.edu
More on Runahead Execution

- Lecture video from Fall 2020, Computer Architecture:
  - https://www.youtube.com/watch?v=zPewo6IaJ_8

- Lecture video from Fall 2017, Computer Architecture:
  - https://www.youtube.com/watch?v=Kj3relihGF4

- Onur Mutlu, "Efficient Runahead Execution Processors"

  Nominated for the ACM Doctoral Dissertation Award by the University of Texas at Austin.
More on Continuous Runahead

- Milad Hashemi, Onur Mutlu, and Yale N. Patt,
  "Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads"
  Proceedings of the 49th International Symposium on Microarchitecture (MICRO), Taipei, Taiwan, October 2016.
  [Slides (pptx) (pdf)] [Lightning Session Slides (pdf)] [Poster (pptx) (pdf)]
  Best paper session.

Continuous Runahead: Transparent Hardware Acceleration for Memory Intensive Workloads

  Milad Hashemi*, Onur Mutlu§, Yale N. Patt*

  *The University of Texas at Austin  §ETH Zürich
Effective prefetching can both improve performance and reduce hardware cost.
HIGH-PERFORMANCE THROUGHPUT COMPUTING

Throughput computing, achieved through multithreading and multicore technology, can lead to performance improvements that are 10 to 30× those of conventional processors and systems. However, such systems should also offer good single-thread performance. Here, the authors show that hardware scouting increases the performance of an already robust core by up to 40 percent for commercial benchmarks.

More on Runahead in Sun ROCK

Simultaneous Speculative Threading: A Novel Pipeline Architecture Implemented in Sun’s ROCK Processor

Shailender Chaudhry, Robert Cypher, Magnus Ekman, Martin Karlsson, Anders Landin, Sherman Yip, Håkan Zeffer, and Marc Tremblay
Sun Microsystems, Inc.
4180 Network Circle, Mailstop SCA18-211
Santa Clara, CA 95054, USA
{shailender.chaudhry, robert.cypher, magnus.ekman, martin.karlsson, anders.landin, sherman.yip, haakan.zeffer, marc.tremblay}@sun.com

Runahead Execution in IBM POWER6

Runahead Execution vs. Conventional Data Prefetching in the IBM POWER6 Microprocessor

Harold W. Cain        Priya Nagpurkar

IBM T.J. Watson Research Center
Yorktown Heights, NY
{tcain, pnagpurkar}@us.ibm.com

DENVER: NVIDIA’S FIRST 64-BIT ARM PROCESSOR

NVIDIA’s first 64-bit ARM processor, code-named Denver, leverages a host of new technologies, such as dynamic code optimization, to enable high-performance mobile computing. Implemented in a 28-nm process, the Denver CPU can attain clock speeds of up to 2.5 GHz. This article outlines the Denver architecture, describes its technological innovations, and provides relevant comparisons against competing mobile processors.

Reducing the effects of long cache-miss penalties has been a major focus of the microarchitecture, using techniques like prefetching and run-ahead. An aggressive hardware prefetcher implementation detects L2 cache requests and tracks up to 32 streams, each with complex stride patterns.

Run-ahead uses the idle time that a CPU spends waiting on a long latency operation to discover cache and DTLB misses further down the instruction stream and generates prefetch requests for these misses. These prefetch requests warm up the data cache and DTLB well before the actual execution of the instructions that require the data. Run-ahead complements the hardware prefetcher because it's better at prefetching nonstrided streams, and it trains the hardware prefetcher faster than normal execution to yield a combined benefit of 13 percent on SPECint2000 and up to 60 percent on SPECfp2000.

The core includes a hardware prefetch unit that Boggs describes as “aggressive” in preloading the data cache but less aggressive in preloading the instruction cache. It also implements a “run-ahead” feature that continues to execute microcode speculatively after a data-cache miss; this execution can trigger additional cache misses that resolve in the shadow of the first miss. Once the data from the original miss returns, the results of this speculative execution are discarded and execution restarts with the bundle containing the original miss, but run-ahead can preload subsequent data into the cache, thus avoiding a string of time-wasting cache misses. These and other features help Denver out-score Cortex-A15 by more than 2.6x on a memory-read test even when both use the same SoC framework (Tegra K1).

Figure 3. Denver CPU microarchitecture. This design combines a fairly


Gwennap, “NVIDIA’s First CPU is a Winner,” MPR 2014.
Looking to the Past
At the Time... Early 2000s...

- Large focus on increasing the size of the window...
  - And, designing bigger, more complicated machines

- Runahead was a different way of thinking
  - Keep the OoO core simple and small
  - At the expense of some benefits (e.g., non-memory-related)
  - Use aggressive “automatic speculative execution” solely for prefetching
  - Synergistic with prefetching and branch prediction methods

- A lot of interesting and innovative ideas ensued...
Improving Data Cache Performance by Pre-executing Instructions Under a Cache Miss

James Dundas and Trevor Mudge
Department of Electrical Engineering and Computer Science
The University of Michigan
Ann Arbor, Michigan 48109-2122
{dundas, tnm}@eecs.umich.edu

Abstract
In this paper we propose and evaluate a technique that improves first level data cache performance by pre-executing future instructions under a data cache miss. We show that these pre-executed instructions can generate highly accurate data prefetches, particularly when the first level cache is small. The technique is referred to as runahead processing. The hardware required to implement runahead is modest, because, when a miss occurs, it makes use of an otherwise idle resource, the execution logic. The principal hardware cost is an extra register file. To measure the impact of runahead, we simulated a processor executing five integer Spec95 benchmarks. Our results show that runahead was able to significantly reduce data cache CPI for four of the five benchmarks. We also compared runahead to a simple form of prefetching, sequential prefetching, which would seem to be suitable for scientific benchmarks. We confirm this by enlarging the scope of our experiments to include a scientific benchmark. However, we show that runahead was also able to outperform sequential prefetching on the scientific benchmark. We also conduct studies that demonstrate that runahead can generate many useful prefetches for lines that show little spatial locality with the misses that initiate runahead episodes. Finally, we discuss some further enhancements of our baseline runahead prefetching scheme.

are allocated by the software. This hybrid hardware-software technique was presented in [8]. Their instruction stride table (IST) selectively generates cache miss initiated prefetches for accesses chosen beforehand by the compiler. This resulted in multiprocessor performance for scientific benchmarks comparable in some cases to software prefetching, with an instruction stride table as small as 4 entries. The IST concept was subsequently combined with the prefetch predicates of [2] in [9]. Another hardware prefetching scheme that avoids the need for significant amounts of hardware is the “wrong path” prefetching described in [10]. This actually prefetches instructions from the not-taken path, in the expectation that they will be executed during a later iteration.

Most prefetching techniques, software- or hardware-based, tend to perform poorly on an important class of applications having recursive data structures such as linked-lists. A software technique that overcomes this limitation was presented recently in [11], in which software prefetches were inserted at subroutine call sites that passed pointers as arguments. Another pointer-based approach was described in [12]. This approach uses pointers stored within the data structures to generate software prefetches.

The runahead prefetching approach presented in this paper is a hardware approach, that requires only a modest amount of hardware, because, when a miss occurs, it makes use of an otherwise
MLP yes! ILP no!
Memory Level Parallelism, or why I no longer care about Instruction Level Parallelism

Andrew Glew
Intel Microcomputer Research Labs and University of Wisconsin, Madison

Problem Description: It should be well known that processors are outstripping memory performance; specifically that memory latencies are not improving as fast as processor cycle time or IPC or memory bandwidth.

Thought experiment: imagine that a cache miss takes 10000 cycles to execute. For such a processor instruction level parallelism is useless, because most of the time is spent waiting for memory. Branch prediction is also less effective, since most branches can be determined with data already in registers or in the cache; branch prediction only helps for branches which depend on outstanding cache misses.

At the same time, pressures for reduced power consumption mount.

Given such trends, some computer architects in industry (although not Intel EPIC) are talking seriously about retreating from out-of-order superscalar processor architecture, and instead building simpler, faster, dumber, 1-wide in-order processors with high degrees of speculation. Sometimes this is proposed in combination with multiprocessing and multithreading: tolerate long memory latencies by switching to other processes or threads.

I propose something different: build narrow fast machines but use intelligent logic inside the CPU to increase the number of outstanding cache misses that can be generated from a single program.

Solution: First, change the mindset: MLP, Memory Level Parallelism, is what matters, not ILP, Instruction Level Parallelism.

By MLP I mean simply the number of outstanding cache misses that can be generated (by a single thread, task, or program) and executed in an overlapped manner. It does not matter what sort of execution engine generates the multiple outstanding cache misses. An out-of-order superscalar ILP CPU may generate multiple outstanding cache misses, but 1-wide processors can be just as effective.

Change the metrics: total execution time remains the overall goal, but instead of reporting IPC as an approximation to this, we must report MLP. Limit studies should be in terms of total number of non-overlapped cache misses on critical path.

Now do the research: Many present-day hot topics in computer architecture help MLP, but do not help MLP. As mentioned above, predicting branch directions for branches that can be determined from data already in the cache or in registers does not help MLP for extremely long latencies. Similarly, prefetching of data cache misses for array processing codes does not help MLP – it just

Instead, investigate microarchitectures that help MLP:

(0) Trivial case – explicit multithreading, like SMT.
(1) Slightly less trivial case – implicitly multithread single programs, either by compiler software on an MT machine, or by a hybrid, such as Wisconsin Multiscalar, or entirely in hardware, as in Intel’s Dynamic Multi-Thread.
(2) Build 1-wide processors that are as fast as possible: use circuit tricks, as well as logic tricks such as redundant encoding for numeric computation and memory addressing.
(3) Allow the hardware dynamic scheduling mechanisms to use sequential algorithms implemented by this narrow, fast, processor, rather than limiting it to parallel algorithms implementable in associative logic.
(4) Build very large instruction windows allowing speculation tens of thousands of instructions ahead. Avoid circuit speed issues by caching the instruction window. Remove small arbitrary limits on the number of cache misses outstanding allowed.
(5) Further reduce the cost of very large instruction windows by throwing away anything that can be recomputed based on data in registers or cache.
(6) Don’t stall speculation because the oldest instruction in the machine is a cache miss. Let the front of the machine continue executing branches, forgetting data dependent on cache misses.
(7) Parallelize linked data structure traversals by building skip lists in hardware – converting sequential data structures into parallel ones. Store these extra skip pointers in main memory.

Call such a processor microarchitecture a “Super-non-blocking” microarchitecture.

Justification: The processor industry (and I am well known) incorrectly optimal cache sizes show only limited headroom. Barring a revolution in memory technology, the Memory Wall is real, and getting closer. Multithreading and multiprocessing have some hope of tolerating memory latency, but only if there are parallel workloads. If single thread performance is still an issue, the only potentially MLP enhancing technologies are what I describe here, or data value prediction – and data value prediction seems to only do well for stuff that fits in the cache.

“Super-non-blocking” processors extends dynamic, out-of-order, execution to maximize MLP, but simplifies it by discarding superscalar ILP as unnecessary.
Looking to the Future
A Look into the Future…

- Microarchitecture (especially memory) is critically important
  - And, fun...
  - And, impactful...

- Runahead is a great example of harmonious industry-academia collaboration

- Fundamental problems will remain fundamental
  - And will require fundamental (and creative) solutions
Citation for the Test of Time Award

- Runahead Execution is a pioneering paper that opened up new avenues in dynamic prefetching.
- The basic idea of runahead execution effectively increases the instruction window very significantly, without having to increase physical resource size (e.g. the issue queue).
- This seminal paper spawned off a new area of ILP-enhancing microarchitecture research.
- This work has had strong industry impact as evidenced by IBM's POWER6 - Load Lookahead, NVIDIA Denver, and Sun ROCK's hardware scouting.
More on Runahead Execution

- Lecture video from Fall 2020, Computer Architecture:
  - https://www.youtube.com/watch?v=zPewo6IaJ_8

- Lecture video from Fall 2017, Computer Architecture:
  - https://www.youtube.com/watch?v=Kj3relihGF4

- Onur Mutlu,
  "Efficient Runahead Execution Processors"
  Nominated for the ACM Doctoral Dissertation Award by the University of Texas at Austin.

https://www.youtube.com/onurmutlulectures
More on Runahead Execution (I)

Review: Runahead Execution (Mutlu et al., HPCA 2003)

Small Window:
- Load 1 Miss
- Load 2 Miss
- Compute
- Stall
- Compute
- Stall
- Miss 1
- Miss 2

Runahead:
- Load 1 Miss
- Load 2 Miss
- Load 1 Hit
- Load 2 Hit
- Compute
- Runahead
- Compute
- Saved Cycles
- 18

Computer Architecture - Lecture 19a: Execution-Based Prefetching (ETH Zürich, Fall 2020)

395 views • Nov 29, 2020

https://www.youtube.com/watch?v=zPewo6laJ_8&list=PL5Q2soXY2Zi9xidylGxBxUz7xRPS-wisBN&index=34
More on Runahead Execution (II)

Runahead Execution in NVIDIA Denver

Reducing the effects of long cache-miss penalties has been a major focus of the microarchitecture, using techniques like prefetching and run-ahead. An aggressive hardware prefetcher implementation detects L2 cache requests and tracks up to 32 streams, each with complex stride patterns.

Run-ahead uses the idle time that a CPU spends waiting on a long latency operation to discover cache and DTLB misses further down the instruction stream and generates prefetch requests for these misses. These prefetch requests warm up the data cache and DTLB well before the actual execution of the instructions that require the data. Run-ahead complements the hardware prefetcher because it’s better at prefetching nonstrided streams, and it trains the hardware prefetcher faster than normal execution to yield a combined benefit of 13 percent on SPECint2000 and up to 60 percent on SPECfp2000.

Gwennap, “NVIDIA's First CPU is a Winner,” MPR 2014.

Onur Mutlu - Runahead Execution: A Short Retrospective (HPCA Test of Time Award Talk @ HPCA 2021)

https://www.youtube.com/watch?v=KFCOecRQTlc
More Recommended Material on Prefetching
Lecture on Prefetching: Fall 2022

Performance with Varying DRAM Bandwidth

- Pythia
- Bingo
- MLOP
- SPP
- Baseline
- ~Intel Xeon 6258R
- ~AMD EPYC Rome 7702P
- ~AMD Threadripper 3990x

Geomean speedup over no prefetching

DRAM MTPS (in log scale)

https://www.youtube.com/watch?v=UjqS9iKo4Ik&list=PL5Q2soXY2Zi-cAls3cyauNzM7-74Eq31O&index=16
Lectures on Prefetching (I)

X86 PREFETCH Instruction

**PREFETCH**—Prefetch Data into Caches

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Instruction</th>
<th>64-Bit Mode</th>
<th>Comp/Log Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0F 18/1</td>
<td>PREFETCH0 m8</td>
<td>Valid</td>
<td>Valid</td>
<td>Move data from m8 closer to the processor using T0 hint.</td>
</tr>
<tr>
<td>0F 18/2</td>
<td>PREFETCH1 m8</td>
<td>Valid</td>
<td>Valid</td>
<td>Move data from m8 closer to the processor using T1 hint.</td>
</tr>
<tr>
<td>0F 18/3</td>
<td>PREFETCH2 m8</td>
<td>Valid</td>
<td>Valid</td>
<td>Move data from m8 closer to the processor using T2 hint.</td>
</tr>
<tr>
<td>0F 18/0</td>
<td>PREFETCHNTA</td>
<td>Valid</td>
<td>Valid</td>
<td>Move data from m8 closer to the processor using NTA hint.</td>
</tr>
</tbody>
</table>

**Description**
Fetched the line of data from memory that contains the byte specified with the source operand and a location in the cache hierarchy specified by a locality hint:
- **T0** (temporal data)—prefetch data into all levels of the cache hierarchy.
  - Pentium III processor—1st- or 2nd-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- **T1** (temporal data with respect to first level cache)—prefetch data into level 2 cache and higher.
  - Pentium III processor—1st-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- **T2** (temporal data with respect to second level cache)—prefetch data into level 2 cache and higher.
  - Pentium III processor—1st-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.
- NTA (non-temporal data with respect to all cache levels)—prefetch data into non-temporal cache structure and into a location close to the processor, minimizing cache pollution.
  - Pentium III processor—1st-level cache.
  - Pentium 4 and Intel Xeon processors—2nd-level cache.

Onur Mutlu Lectures
16.5K subscribers

https://www.youtube.com/watch?v=xZmDyj0g3Pw&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=33
Lectures on Prefetching (II)

Thread-Based Pre-Execution


https://www.youtube.com/watch?v=zPewo6IaJ_8&list=PL5Q2soXY2Zi9xIdyIgBxUz7xRPS-wisBN&index=34
Lectures on Prefetching (III)

Runahead Execution in NVIDIA Denver

Reducing the effects of long cache-miss penalties has been a major focus of the microarchitecture, using techniques like prefetching and run-ahead. An aggressive hardware prefetcher implementation detects L2 cache requests and tracks up to 32 streams, each with complex stride patterns.

Run-ahead uses the idle time that a CPU spends waiting on a long latency operation to discover cache and DTLB misses further down in the instruction stream and generates prefetch requests for these misses. These prefetch requests warm up the data cache and DTLB well before the actual execution of the instructions that require the data. Run-ahead complements the hardware prefetcher because it’s better at prefetching nonstrided streams, and it trains the hardware prefetcher faster than normal execution to yield a combined benefit of 13 percent on SPECint2000 and up to 60 percent on SPECfp2000.


Gwennap, "NVIDIA’s First CPU is a Winner," MPR 2014.

Onur Mutlu - Runahead Execution: A Short Retrospective (HPCA Test of Time Award Talk @ HPCA 2021)

1,162 views • Premiered Mar 6, 2021

https://www.youtube.com/watch?v=KFCOecRQTlc
Lectures on Prefetching (IV)

Software Prefetching (II)

for (i=0; i<N; i++) {
    __prefetch(a[i+8]);
    __prefetch(b[i+8]);
    sum += a[i]*b[i];
}

while (p) {
    __prefetch(p->next);
    work(p->data);
    p = p->next;
}

while (p) {
    __prefetch(p->next->next->next);
    work(p->data);
    p = p->next;
}

Which one is better?

Can work for very regular array-based access patterns. Issues:

- Prefetch instructions take up processing/execution bandwidth
- How early to prefetch? Determining this is difficult
  - Prefetch distance depends on hardware implementation (memory latency, cache size, time between loop iterations) → portability?
  - Going too far back in code reduces accuracy (branches in between)
- Need “special” prefetch instructions in ISA?
  - Alpha load into register 31 treated as prefetch (r31==0)
  - PowerPC dcbt (data cache block touch) instruction
- Not easy to do for pointer-based data structures


5,216 views • Apr 3, 2015

https://www.youtube.com/watch?v=ibPL7T9iEwY&list=PL5PHm2jkkXmi5Cxxl7b3JCL1TWybTDtKq&index=29
Lecture 26. More Prefetching and Emerging Memory Technologies - CMU - Comp. Arch. 2015 - Onur Mutlu

3,642 views • Apr 6, 2015

Address Correlation Based Prefetching (II)

Idea: Record the likely-next addresses (B, C, D) after seeing an address A
- Next time A is accessed, prefetch B, C, D
- A is said to be correlated with B, C, D

Prefetch up to N next addresses to increase coverage
- Prefetch accuracy can be improved by using multiple addresses as key for the next address: (A, B) → (C)
- (A, B) correlated with C

Lectures on Prefetching

- Computer Architecture, Fall 2020, Lecture 18
  - Prefetching (ETH, Fall 2020)
  - https://www.youtube.com/watch?v=xZmDyj0g3Pw&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=33

- Computer Architecture, Fall 2020, Lecture 19a
  - Execution-Based Prefetching (ETH, Fall 2020)
  - https://www.youtube.com/watch?v=zPewo6IaJ_8&list=PL5Q2soXY2Zi9xidyIgBxUz7xRPS-wisBN&index=34

- Computer Architecture, Spring 2015, Lecture 25
  - Prefetching (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=ibPL7T9iEwY&list=PL5PHm2jkkXmi5CxxI7b3JC_L1TWybTDtKq&index=29

- Computer Architecture, Spring 2015, Lecture 26
  - More Prefetching (CMU, Spring 2015)
  - https://www.youtube.com/watch?v=TUFins4z6o4&list=PL5PHm2jkkXmi5CxxI7b3JC_L1TWybTDtKq&index=30

https://www.youtube.com/onurmutlulectures
Research Opportunities
Computer Architecture Research

- **If you want to do research** in any of the covered topics or any topic in Comp Arch, HW/SW Interaction & related areas
  - We have many projects and a great environment to perform top-notch research, bachelor’s/master’s/semester projects
  - Talk with me (email, whatsapp, etc.) & apply online

- **Many research topics and projects**
  - Memory (DRAM, NVM, Flash, SW/HW issues, emerging tech)
  - Processing in Memory
  - Hardware Security
  - New Computing Paradigms
  - Machine Learning for System Design
  - System Design for AI/ML, Health, Genomics, Medicine
  - ...

A lot of room for creativity, innovation & high impact
Current Research Mission

Computer architecture, HW/SW, systems, bioinformatics, security

Heterogeneous Processors and Accelerators

Hybrid Main Memory

Persistent Memory/Storage

Graphics and Vision Processing

Build fundamentally better architectures
The Transformation Hierarchy

Computer Architecture (expanded view)
- Problem
- Algorithm
- Program/Language
- System Software
- SW/HW Interface
- Micro-architecture
- Logic
- Devices
- Electrons

Computer Architecture (narrow view)
SAFARI Research Mission & Major Topics

Build fundamentally better architectures

- Data-centric systems: memory/storage systems
  - Proc. in Memory/Storage, emerging tech, DRAM
-Fundamentally secure/reliable/safe architectures
  - RowHammer; patchable HW; secure memory
- Low-latency & predictable architectures
  - Low-latency, low-energy yet low-cost memory
  - QoS-aware and predictable memory systems
- Systems for ML/AI/Genomics/Health/Graphs
  - Algorithm/architecture co-design; accelerators
- Data-driven and data-aware architectures
  - ML/AI for architectural control and design
  - Expressive memory and expressive systems
- Ultra-fast & efficient genome analysis

Problem
Algorithm
Program/Language
System Software
SW/HW Interface
Micro-architecture
Logic
Devices
Electrons

Broad research spanning apps, systems, logic with architecture at the center

SAFARI
https://safari.ethz.ch
https://people.inf.ethz.ch/omutlu/projects.htm

"SAFARI Research Group: Introduction & Research"
[Slides (pptx) (pdf)] [Talk Video (15 min.)]
Open Source Tools: SAFARI GitHub

SAFARI Research Group at ETH Zurich and Carnegie Mellon University
Site for source code and tools distribution from SAFARI Research Group at ETH Zurich and Carnegie Mellon University.

- ETH Zurich and Carnegie Mellon U...
- https://safari.ethz.ch/
- omutlu@gmail.com

Pinned

- **ramulator**  Public
  A Fast and Extensible DRAM Simulator, with built-in support for modeling many different DRAM technologies including DDRx, LPDDRx, GDDRx, WIOx, HBMx, and various academic proposals. Described in the...
  - C++  311  161

- **prim-benchmarks**  Public
  PriM (Processing-In-Memory benchmarks) is the first benchmark suite for a real-world processing-in-memory (PIM) architecture. PriM is developed to evaluate, analyze, and characterize the first publi...
  - C  53  21

- **DAMOV**  Public
  DAMOV is a benchmark suite and a methodical framework targeting the study of data movement bottlenecks in modern applications. It is intended to study new architectures, such as near-data processin...
  - C++  26  4

- **SneakySnake**  Public
  SneakySnake is the first and the only pre-alignment filtering algorithm that works efficiently and fast on modern CPU, FPGA, and GPU architectures. It greatly (by more than two orders of magnitude...
  - VHDL  41  8

- **MQSim**  Public
  MQSim is a fast and accurate simulator modeling the performance of modern multi-queue (MQ) SSDs as well as traditional SATA based SSDs. MQSim faithfully models new high-bandwidth protocol implement...
  - C++  146  93

- **rowhammer**  Public
  - C  189  41

https://github.com/CMU-SAFARI/
Onur Mutlu’s SAFARI Research Group

Computer architecture, HW/SW, systems, bioinformatics, security, memory

https://safari.ethz.ch/safari-newsletter-april-2020/

Think BIG, Aim HIGH!

SAFARI
https://safari.ethz.ch
Dear SAFARI friends,

Happy New Year! We are excited to share our group highlights with you in this second edition of the SAFARI newsletter (You can find the first edition from April 2020 here). 2020 has
SAFARI Newsletter December 2021 Edition

https://safari.ethz.ch/safari-newsletter-december-2021/
A Talk on Our Research & Teaching

Applying to Grad School & Doing Impactful Research

Onur Mutlu
omutlu@gmail.com
https://people.inf.ethz.ch/omutlu
13 June 2020
Undergraduate Architecture Mentoring Workshop @ ISCA 2021

SAFARI ETH Zürich Carnegie Mellon

Arch. Mentoring Workshop @ISCA’21 - Applying to Grad School & Doing Impactful Research - Onur Mutlu
1,563 views • Premiered Jun 16, 2021

Onur Mutlu Lectures
17.2K subscribers

Panel talk at Undergraduate Architecture Mentoring Workshop at ISCA 2021
(https://sites.google.com/wisc.edu/uar...)

https://www.youtube.com/watch?v=83tlorht7Mc&list=PL5Q2soXY2Zi8D_5MGV6EnXEJHnV2YFBJl&index=54
An Interview on Computing Futures

https://www.youtube.com/watch?v=8ffSEKZhmvo
Computer Architecture Research

- **If you want to do research** in any of the covered topics or any topic in Comp Arch, HW/SW Interaction & related areas
  - We have many projects and a great environment to perform top-notch research, bachelor’s/master’s/semester projects
  - Talk with me (email, whatsapp, etc.) & apply online

- **Many research topics and projects**
  - Memory (DRAM, NVM, Flash, SW/HW issues, emerging tech)
  - Processing in Memory
  - Hardware Security
  - New Computing Paradigms
  - Machine Learning for System Design
  - System Design for AI/ML, Health, Genomics, Medicine
  - ...

A lot of room for creativity, innovation & high impact
Multiprocessors
Readings: Multiprocessing

- **Required**

- **Recommended**
Memory Consistency

- Required
Readings: Cache Coherence

- **Required**

- **Recommended:**
  - Culler and Singh, *Parallel Computer Architecture*
    - Chapter 5.1 (pp 269 – 283), Chapter 5.3 (pp 291 – 305)
  - P&H, *Computer Organization and Design*
    - Chapter 5.8 (pp 534 – 538 in 4th and 4th revised eds.)
Multiprocessors and Issues in Multiprocessing
Flynn’s Taxonomy of Computers


- **SISD**: Single instruction operates on single data element
- **SIMD**: Single instruction operates on multiple data elements
  - Array processor
  - Vector processor
- **MISD**: Multiple instructions operate on single data element
  - Closest form: systolic array processor, streaming processor
- **MIMD**: Multiple instructions operate on multiple data elements (multiple instruction streams)
  - Multiprocessor
  - Multithreaded processor
SIMD Example: Vector & Array Processors

Array vs. Vector Processors

**ARRAY PROCESSOR**
- PE0
- PE1
- PE2
- PE3

**VECTOR PROCESSOR**
- LD
- ADD
- MUL
- ST

Instruction Stream
- LD  VR ← A[3:0]
- ADD VR ← VR, 1
- MUL VR ← VR, 2
- ST  A[3:0] ← VR

**Same op @ same time**
- LD0
- LD1
- LD2
- LD3
- AD0
- AD1
- AD2
- AD3
- MU0
- MU1
- MU2
- MU3
- ST0
- ST1
- ST2
- ST3

**Different ops @ same space**
- LD0
- LD1
- LD2
- LD3
- AD0
- AD1
- MU0
- AD2
- MU1
- ST0
- AD3
- MU2
- ST1
- MU3
- ST2
- ST3

**Same op @ space**
- ST3

Livestream - Digital Design and Computer Architecture - ETH Zürich (Spring 2022)
Digital Design & Computer Arch. - Lecture 20: SIMD Processing (Vector and Array Processors) (S 2022)

https://www.youtube.com/watch?v=YPLPVadgw-E&list=PL5Q2soXY2Zi97Ya5DEUpMpO2bbAoaG7c6&index=23
MISD Example: Systolic Arrays

An Example Modern Systolic Array: TPU (II)

As reading a large SRAM uses much more power than arithmetic, the matrix unit uses systolic execution to save energy by reducing reads and writes of the Unified Buffer [Kun80][Ram91][Ovt15b]. Figure 4 shows that data flows in from the left, and the weights are loaded from the top. A given 256-element multiply-accumulate operation moves through the matrix as a diagonal wavefront. The weights are preloaded, and take effect with the advancing wave alongside the first data of a new block. Control and data are pipelined to give the illusion that the 256 inputs are read at once, and that they instantly update one location of each of 256 accumulators. From a correctness perspective, software is unaware of the systolic nature of the matrix unit, but for performance, it does worry about the latency of the unit.


Digital Design & Computer Arch. - Lecture 19: VLIW and Systolic Array Architectures (Spring 2022)
842 views - Premiered May 6, 2022

Onur Mutlu Lectures
24.5K subscribers

Digital Design and Computer Architecture, ETH Zürich, Spring 2022 (https://safari.ethz.ch/digitalarchitektur/)

Lecture 19a: VLIW Architectures
Lecture 19b: Systolic Array Architectures
Lecturer: Professor Onur Mutlu (https://people.inf.ethz.ch/omutlu/)
Date: May 6, 2022

https://youtu.be/1SSqV7Y75oU?t=2316
Why Parallel Computers?

- Parallelism: Doing multiple things at a time
- Things: instructions, operations, tasks

Main (or Original) Goal
- Improve performance (Execution time or task throughput)
  - Execution time of a program governed by Amdahl’s Law

Other Goals
- Reduce power consumption
  - (4N units at freq F/4) consume less power than (N units at freq F)
  - Why?
- Improve cost efficiency and scalability, reduce complexity
  - Harder to design a single unit that performs as well as N simpler units
- Improve dependability: Redundant execution in space
Types of Parallelism and How to Exploit Them

- **Instruction Level Parallelism**
  - Different instructions within a stream can be executed in parallel
  - Pipelining, out-of-order execution, speculative execution, VLIW
  - Dataflow

- **Data Parallelism**
  - Different pieces of data can be operated on in parallel
  - SIMD: Vector processing, array processing
  - Systolic arrays, streaming processors

- **Task Level Parallelism**
  - Different “tasks/threads” can be executed in parallel
  - Multithreading
  - Multiprocessing (multi-core)
Task-Level Parallelism: Creating Tasks

- Partition a single problem into multiple related tasks (threads)
  - Explicitly: Parallel programming
    - Easy when tasks are natural in the problem
      - Web/database queries
    - Difficult when natural task boundaries are unclear
  - Transparently/implicitly: Thread level speculation
    - Partition a single thread speculatively

- Run many independent tasks (processes) together
  - Easy when there are many processes
    - Batch simulations, different users, cloud computing workloads
  - Does not improve the performance of a single task
Multiprocessing Fundamentals
Multiprocessor Types

- Loosely coupled multiprocessors
  - No shared global memory address space
  - Multicomputer network
    - Network-based multiprocessors
  - Usually programmed via message passing
    - Explicit calls (send, receive) for communication

- Tightly coupled multiprocessors
  - Shared global memory address space
  - Traditional multiprocessing: symmetric multiprocessing (SMP)
    - Existing multi-core processors, multithreaded processors
  - Programming model similar to uniprocessors (i.e., multitasking uniprocessor) except
    - Operations on shared data require synchronization
Main Design Issues in Tightly-Coupled MP

- **Shared memory synchronization**
  - How to handle synchronization: locks, atomic operations, barriers

- **Cache coherence**
  - How to ensure correct operation in the presence of private caches keeping the same memory address cached

- **Memory consistency: Ordering of all memory operations**
  - What should the programmer expect the hardware to provide?

- **Shared resource management**

- **Communication: Interconnects**
Main Programming Issues in Tightly-Coupled MP

- **Load imbalance**
  - How to partition a single task into multiple tasks

- **Synchronization**
  - How to synchronize (efficiently) between tasks
  - How to communicate between tasks
  - Locks, barriers, pipeline stages, condition variables, semaphores, atomic operations, ...

- **Contention (avoidance & management)**
- **Maximizing parallelism**
- **Ensuring correct operation while optimizing for performance**
Aside: Hardware-based Multithreading

- Coarse grained
  - Quantum based
  - Event based (switch-on-event multithreading), e.g., switch on L3 miss

- Fine grained
  - Cycle by cycle

- Simultaneous
  - Can dispatch instructions from multiple threads at the same time
  - Good for improving execution unit utilization
Fine-Grained Multithreading

Idea: Hardware has multiple thread contexts (PC+registers). Each cycle, fetch engine fetches from a different thread.

- By the time the fetched branch/instruction resolves, no instruction is fetched from the same thread
- Branch/instruction resolution latency overlapped with execution of other threads’ instructions

+ No logic needed for handling control and data dependences within a thread
  -- Single thread performance suffers
  -- Extra logic for keeping thread contexts
  -- Does not overlap latency if not enough threads to cover the whole pipeline
More on Multithreading (II)

Carnegie Mellon - Parallel Computer Architecture 2012 - Onur Mutlu - Lecture 10 - Multithreading II

Lecture 10: Multithreading II
Lecturer: Prof. Onur Mutlu (http://users.ece.cmu.edu/~omutlu/)
Date: September 28, 2012.

https://www.youtube.com/onurmutlulectures
More on Multithreading (III)

https://www.youtube.com/onurmutlulectures
Lectures on Multithreading

- Parallel Computer Architecture, Fall 2012, Lecture 9
  - Multithreading I (CMU, Fall 2012)
    - https://www.youtube.com/watch?v=iqi9wFqFiNU&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnyV6D&index=51

- Parallel Computer Architecture, Fall 2012, Lecture 10
  - Multithreading II (CMU, Fall 2012)
    - https://www.youtube.com/watch?v=e8lfl6MbILq&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnyV6D&index=52

- Parallel Computer Architecture, Fall 2012, Lecture 13
  - Multithreading III (CMU, Fall 2012)
    - https://www.youtube.com/watch?v=7vkDpZ1-hHM&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnyV6D&index=53

- Parallel Computer Architecture, Fall 2012, Lecture 15
  - Speculation I (CMU, Fall 2012)
    - https://www.youtube.com/watch?v=-hbmzIDe0sA&list=PL5PHm2jkkXmgDN1PLwOY_tGtUlynnyV6D&index=54

https://www.youtube.com/onurmutlulectures
Limits of Parallel Speedup
Parallel Speedup Example

- \( a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \)

- Assume given inputs: \( x \) and each \( a_i \)

- Assume each operation 1 cycle, no communication cost, each operation can be executed in a different processor

- How fast is this with a single processor?
  - Assume no pipelining or concurrent execution of instructions

- How fast is this with 3 processors?
\[ R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0 \]

Single processor: 11 operations (data flow graph)

\[ T_1 = 11 \text{ cycles} \]
$R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$

Three processors: $T_3$ (execution with 3 proc.)

$T_3 = 5\text{ cycles}$
Speedup with 3 Processors

\[ T_3 = \frac{5}{5} \text{ cycles} \]

Speedup with 3 processors = \[ \frac{11}{5} = 2.2 \]

\[ \left( \frac{T_1}{T_3} \right) \]

Is this a fair comparison?
Revisiting the Single-Processor Algorithm

Revisit $T_1$

Better single-processor algorithm:

$$R = a_4 x^4 + a_3 x^3 + a_2 x^2 + a_1 x + a_0$$

$$R = (((a_4 x + a_3) x + a_2) x + a_1) x + a_0$$

(Horner's method)

\[ T_1 = 8 \text{ cycles} \]

**Speedup with 3 proc.**

\[
\frac{T_1^{\text{best}}}{T_3^{\text{best}}} = \frac{8}{5} = 1.6
\]  

(ncf 2.2)
Superlinear Speedup

- Can speedup be greater than P with P processing elements?

- **Unfair comparisons**
  Compare best parallel algorithm to wimpy serial algorithm $\rightarrow$ unfair

- **Cache/memory effects**
  More processors $\rightarrow$ more cache or memory $\rightarrow$ fewer misses in cache/mem
Utilization, Redundancy, Efficiency

- Traditional metrics
  - Assume all $P$ processors are tied up for parallel computation

- Utilization: How much processing capability is used
  - $U = \frac{\text{(# Operations in parallel version)}}{\text{processors} \times \text{Time}}$

- Redundancy: how much extra work is done with parallel processing
  - $R = \frac{\text{(# of operations in parallel version)}}{\text{(# operations in best single processor algorithm version)}}$

- Efficiency
  - $E = \frac{\text{(Time with 1 processor)}}{\text{(processors} \times \text{Time with P processors)}}$
  - $E = \frac{U}{R}$
Utilization of a Multiprocessor

**Multiprocessor Metrics**

**Utilization**: How much processing capability we use

<table>
<thead>
<tr>
<th></th>
<th>(P)</th>
<th>(\pi)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

\[ U = \frac{\text{Ops with } P \text{ proc.}}{P \times \pi} \]

\[ U = \frac{10 \text{ operations (in parallel version)}}{3 \text{ processors } \times 5 \text{ time units}} = \frac{10}{15} \]
Redundancy: How much extra work due to multiprocessing

\[ R = \frac{\text{Ops with } p \text{ proc. best}}{\text{Ops with 1 proc. best}} = \frac{10}{8} \]

\[ R \text{ is always } \geq 1 \]

Efficiency: How much resource we use compared to how much resource we can get away with

\[ E = \frac{1}{\frac{T_1^\text{best}}{p \cdot T_p^\text{best}}} \]

\[ (\text{tying up } 1 \text{ proc. for } T_p \text{ time units}) \]

\[ = \frac{8}{15} \]

\[ (E = \frac{U}{R}) \]
Amdahl’s Law and Caveats of Parallelism
Amdahl’s Law

- Amdahl’s Law
  - \( f \): Parallelizable fraction of a program
  - \( N \): Number of processors

\[
\text{Speedup} = \frac{1}{1 - f + \frac{f}{N}}
\]


- Maximum speedup limited by serial portion: Serial bottleneck
Caveats of Parallelism (I)

Why the reality? (diminishing returns)

\[ T_p = \alpha \cdot \frac{T_1}{p} + (1-\alpha) \cdot T_1 \]

parallelizable part/portion of the single-processor program

non-parallelizable part
Amdahl’s Law

\[
\text{Speedup} = \frac{T_1}{T_p} = \frac{1}{\alpha + (1-\alpha) \frac{P}{p}}
\]

as \( p \to \infty \)

Amdahl’s Law Implication 1

Amdahl’s Law illustrated

Adding more and more processors gives less and less benefit if $\alpha < 1$
Amdahl’s Law Implication 2

The benefit (speedup) is small until $\alpha \approx 1$. 

[Graph showing speedup vs. $\alpha$ with curves $P_1$, $P_2$, and $P_3$.]
Caveats of Parallelism (II)

- Amdahl’s Law
  - $f$: Parallelizable fraction of a program
  - $N$: Number of processors

\[
\text{Speedup} = \frac{1}{1 - f} + \frac{f}{N}
\]

- Maximum speedup limited by serial portion: **Serial bottleneck**
- Parallel portion is usually not perfectly parallel
  - **Synchronization** overhead (e.g., updates to shared data)
  - **Load imbalance** overhead (imperfect parallelization)
  - **Resource sharing** overhead (contention among $N$ processors)
Sequential Bottleneck

![Graph showing speedup vs. parallel fraction for different values of N (10, 100, 1000)].
Why the Sequential Bottleneck?

- Parallel machines have the sequential bottleneck

- Main cause: Non-parallelizable operations on data (e.g. non-parallelizable loops)
  
  \[
  \text{for ( } i = 0 ; i < N; i++) \\
  \]

- There are other causes as well:
  - Single thread prepares data and spawns parallel tasks (usually sequential)
Another Example of Sequential Bottleneck (I)

InitPriorityQueue(PQ);
SpawnThreads();
ForEach Thread:

while (problem not solved)
    Lock (X)
    SubProblem = PQ.remove();
    Unlock(X);
    Solve(SubProblem);
    If(problem solved) break;
    NewSubProblems = Partition(SubProblem);
    Lock(X)
    PQ.insert(NewSubProblems);
    Unlock(X)
... 

PrintSolution();

Suleman+,”Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures,” ASPLOS 2009.
Another Example of Sequential Bottleneck (II)

Bottlenecks in Parallel Portion

- **Synchronization:** Operations manipulating shared data cannot be parallelized
  - Locks, mutual exclusion, barrier synchronization
  - **Communication:** Tasks may need values from each other
    - Causes thread serialization when shared data is contended

- **Load Imbalance:** Parallel tasks may have different lengths
  - Due to imperfect parallelization or microarchitectural effects
    - Reduces speedup in parallel portion

- **Resource Contention:** Parallel tasks can share hardware resources, delaying each other
  - Replicating all resources (e.g., memory) expensive
    - Additional latency not present when each task runs alone
Bottlenecks in Parallel Portion: Another View

- Threads in a multi-threaded application can be inter-dependent
  - As opposed to threads from different applications

- Such threads can synchronize with each other
  - Locks, barriers, pipeline stages, condition variables, semaphores, ...

- Some threads can be on the critical path of execution due to synchronization; some threads are not

- Within a thread, some “code segments” may be on the critical path of execution; some are not
Remember: Critical Sections

- Enforce mutually exclusive access to shared data
- Only one thread can be executing it at a time
- Contended critical sections make threads wait → threads causing serialization can be on the critical path

Each thread:

```
loop {
    Compute
    lock(A)
    Update shared data
    unlock(A)
}
```
Critical Section Example from MySQL

Critical Section

Open database tables

Perform the operations

....

Access Open Tables Cache

Parallel

Asymmetric

Symmetric

Chip Area (cores)

Speedup
Remember: Barriers

- Synchronization point
- Threads have to wait until all threads reach the barrier
- Last thread arriving to the barrier is on the critical path

Each thread:

```c
loop1 {
    Compute
}
barrier

loop2 {
    Compute
}
```
Remember: Stages of Pipelined Programs

- Loop iterations are statically divided into code segments called *stages*.
- Threads execute stages on different cores.
- Thread executing the slowest stage is on the critical path.

```plaintext
loop {
    Compute1
    Compute2
    Compute3
}
```
Difficulty in Parallel Programming

- Little difficulty if parallelism is natural
  - “Embarrassingly parallel” applications
  - Multimedia, physical simulation, graphics
  - Large web servers, databases?

- Difficulty is in
  - Getting parallel programs to work correctly
  - Optimizing performance in the presence of bottlenecks

- Much of parallel computer architecture is about
  - Designing machines that overcome the sequential and parallel bottlenecks to achieve higher performance and efficiency
  - Making programmer’s job easier in writing correct and high-performance parallel programs
Some Readings on Bottlenecks & Bottleneck Acceleration
Parallel Application Memory Scheduling

- Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, Onur Mutlu, and Yale N. Patt,

"Parallel Application Memory Scheduling"
Proceedings of the 44th International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, December 2011. Slides (pptx)
Accelerated Critical Sections

- M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt, "Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures"

One of the 13 computer architecture papers of 2009 selected as Top Picks by IEEE Micro.

Accelerating Critical Section Execution with Asymmetric Multi-Core Architectures

M. Aater Suleman
University of Texas at Austin
suleman@hps.utexas.edu

Onur Mutlu
Carnegie Mellon University
onur@cmu.edu

Moinuddin K. Qureshi
IBM Research
mkquresh@us.ibm.com

Yale N. Patt
University of Texas at Austin
patt@ece.utexas.edu
Bottleneck Identification & Scheduling


Bottleneck Identification and Scheduling in Multithreaded Applications

José A. Joao  
ECE Department  
The University of Texas at Austin  
joao@ece.utexas.edu

M. Aater Suleman  
Calxeda Inc.  
aater.suleman@calxeda.com

Onur Mutlu  
Computer Architecture Lab.  
Carnegie Mellon University  
onur@cmu.edu

Yale N. Patt  
ECE Department  
The University of Texas at Austin  
patt@ece.utexas.edu
Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs

Jose A. Joao, M. Aater Suleman, Onur Mutlu, and Yale N. Patt,
"Utility-Based Acceleration of Multithreaded Applications on Asymmetric CMPs"
Proceedings of the 40th International Symposium on Computer Architecture (ISCA), Tel-Aviv, Israel, June 2013. Slides (ppt) Slides (pdf)
Data Marshaling

- M. Aater Suleman, Onur Mutlu, Jose A. Joao, Khubaib, and Yale N. Patt, "Data Marshaling for Multi-core Architectures"


One of the 11 computer architecture papers of 2010 selected as Top Picks by IEEE Micro.

Data Marshaling for Multi-core Architectures

M. Aater Suleman† Onur Mutlu§ José A. Joao† Khubaib† Yale N. Patt†

†The University of Texas at Austin
{suleman, joao, khubaib, patt}@hps.utexas.edu

§Carnegie Mellon University
onur@cmu.edu
Lectures on Bottleneck Acceleration

- Lecture 17b: Parallelism and Heterogeneity
  - Comp Arch, ETH Zurich, Fall 2021
  - https://www.youtube.com/watch?v=GLzG_rEDn9A&list=PL5Q2soXY2Zi-Mnk1PxjEIG32HAGILkTOF&index=18

- Lecture 18a: Bottleneck Acceleration
  - Comp Arch, ETH Zurich, Fall 2021
  - https://www.youtube.com/watch?v=P8l3SMAbyYw&list=PL5Q2soXY2Zi-Mnk1PxjEIG32HAGILkTOF&index=19
Lecture on Parallelism & Heterogeneity

ACMP Performance vs. Parallelism

Area-budget = 16 small cores

<table>
<thead>
<tr>
<th></th>
<th>&quot;Tile-Large&quot;</th>
<th>&quot;Tile-Small&quot;</th>
<th>ACMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Large Cores</td>
<td>4</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Small Cores</td>
<td>0</td>
<td>16</td>
<td>12</td>
</tr>
<tr>
<td>Serial</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Performance</td>
<td>2 x 4 = 8</td>
<td>1 x 16 = 16</td>
<td>1x2 + 1x12 = 14</td>
</tr>
<tr>
<td>Parallel</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Throughput</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

https://www.youtube.com/watch?v=GLzG_rEDn9A&list=PL5Q2soXY2Zi-Mnk1PxjEIG32HAGILkTOF&index=19
Lecture on Bottleneck Acceleration

Bottleneck Acceleration

- Acceleration Index Table (AIT)
  - bid=x4700, large core 0

- Scheduling Buffer (SB)
  - bid=x4600, twc=100
  - bid=x4700, twc=10000

- Bottleneck Table (BT)

- Small Core
  - bid=x4700, large core 0

- Large Core

Livestream · Computer Architecture · ETH Zürich (Fall 2021)

Computer Architecture - Lecture 18: Parallelism & Heterogeneity II (Fall 2021)

Onur Mutlu Lectures
29.2K subscribers

2,058 views · Streamed live on Nov 26, 2021
Computer Architecture, ETH Zürich, Fall 2021 (https://safari.ethz.ch/architecture/...)

https://www.youtube.com/watch?v=P8l3SMAbyYw&list=PL5Q2soXY2Zi-Mnk1PxjEIG32HAGILkTOF&index=19
An Example Parallel Problem: Task Assignment to Processors
Static versus Dynamic Scheduling

- **Static**: Done at compile time or parallel task creation time
  - Schedule does not change based on runtime information

- **Dynamic**: Done at run time (e.g., after tasks are created)
  - Schedule changes based on runtime information

- **Example**: Instruction scheduling
  - Why would you like to do dynamic scheduling?
  - What pieces of information are not available to the static scheduler?
Parallel Task Assignment: Tradeoffs

- Problem: N tasks, P processors, N>P. Do we assign tasks to processors statically (fixed) or dynamically (adaptive)?

- Static assignment
  + Simpler: No movement of tasks.
  - Inefficient: Underutilizes resources when load is not balanced

- Dynamic assignment
  + Efficient: Better utilizes processors when load is not balanced
  - More complex: Need to move tasks to balance processor load
  - Higher overhead: Task movement takes time, can disrupt locality

When can load not be balanced?
Parallel Task Assignment: Example

- Compute histogram of a large set of values
- Parallelization:
  - Divide the values across \( T \) tasks
  - Each task computes a local histogram for its value set
  - Local histograms merged with global histograms in the end

```c
getPageHistogram(Page *P)
{
    For each thread:
    { 
        /* Parallel part of the function */
        UpdateLocalHistogram(Fraction of Page)

        /* Serial part of the function */
        Critical Section:
        Add local histogram to global histogram
        Barrier
    }
    Return global histogram
}
```
Parallel Task Assignment: Example (II)

- How to schedule tasks updating local histograms?
  - Static: Assign equal number of tasks to each processor
  - Dynamic: Assign tasks to a processor that is available
  - When does static work as well as dynamic?

- Implementation of Dynamic Assignment with Task Queues

![Diagram of task assignment](image)
Software Task Queues

- What are the advantages and disadvantages of each?
  - Centralized
  - Distributed
  - Hierarchical

(a) Distributed Task Stealing
(b) Hierarchical Task Queuing
Task Stealing

- **Idea:** When a processor’s task queue is empty it steals a task from another processor’s task queue
  - Whom to steal from? (Randomized stealing works well)
  - How many tasks to steal?

+ Dynamic balancing of computation load

- Additional communication/synchronization overhead between processors
- Need to stop stealing if no tasks to steal
Parallel Task Assignment: Tradeoffs

- Who does the assignment? Hardware versus software?

- Software
  + Better scope
  - More time overhead
  - Slow to adapt to dynamic events (e.g., a processor becoming idle)

- Hardware
  + Low time overhead
  + Can adjust to dynamic events faster
  - Requires hardware changes (area and possibly energy overhead)
How Can the Hardware Help?

- Managing task queues in software has overhead
  - Especially high when task sizes are small

- An idea: Hardware Task Queues
  - Each processor has a dedicated task queue
  - Software fills the task queues (on demand)
  - Hardware manages movement of tasks from queue to queue
  - There can be a global task queue as well → hierarchical tasking in hardware

  - Optional reading
Dynamic Task Generation

- Does static task assignment work in this case?
- Problem: Searching the exit of a maze

```plaintext
while (problem not solved)

  SubProblem = PriorityQ.remove()

  Solve(SubProblem)
  if (solved)
    break
  NewSubProblems = Partition(SubProblem)

  PriorityQ.insert(NewSubProblems)
```
Programming Model vs. Hardware Execution Model
Programming Models vs. Architectures

- Five major models
  - (Sequential)
  - Shared memory
  - Message passing
  - Data parallel (SIMD)
  - Dataflow
  - Systolic

- Hybrid models?
Shared Memory vs. Message Passing

- Are these programming models or execution models supported by the hardware architecture?

- Does a multiprocessor that is programmed by “shared memory programming model” have to support a shared address space processors?

- Does a multiprocessor that is programmed by “message passing programming model” have to have no shared address space between processors?
Programming Models: Message Passing vs. Shared Memory

- **Difference:** how communication is achieved between tasks

  - **Message passing programming model**
    - Explicit communication via messages
    - Loose coupling of program components
    - Analogy: telephone call or letter, no shared location accessible to all

  - **Shared memory programming model**
    - Implicit communication via memory operations (load/store)
    - Tight coupling of program components
    - Analogy: bulletin board, post information at a shared space

- **Suitability of the programming model** depends on the problem to be solved. Issues affected by the model include:
  - Overhead, scalability, ease of programming, bugs, match to underlying hardware, ...
Message Passing vs. Shared Memory Hardware

- Difference: how task communication is supported in hardware

- **Shared memory hardware (or machine model)**
  - All processors see a global shared address space
    - Ability to access all memory from each processor
  - A write to a location is visible to the reads of other processors

- **Message passing hardware (machine model)**
  - No global shared address space
  - Send and receive variants are the only method of communication between processors (much like networks of workstations today, i.e. clusters)

- Suitability of the hardware depends on the problem to be solved as well as the programming model.
Most of parallel computing history, there was no separation between programming model and hardware

- Message passing: Caltech Cosmic Cube, Intel Hypercube, Intel Paragon
- Shared memory: CMU C.mmp, Sequent Balance, SGI Origin.
- SIMD: ILLIAC IV, CM-1

However, any hardware can really support any programming model

Why?
- Application → compiler/library → OS services → hardware