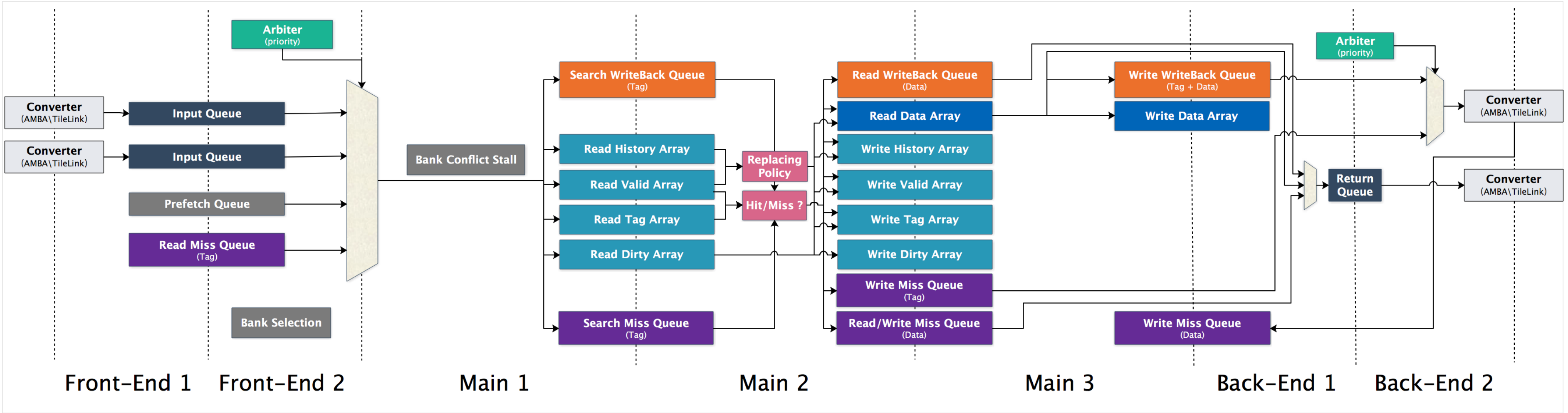


OpenPrefetch

Let There Be Industry-Competitive Prefetching in RISC-V Processors

(in-progress)

- Bowen Huang, Zihao Yu, Zhigang Liu, Chuanqi Zhang, Sa Wang, Yungang Bao
  - Institute of Computing Technology(ICT), Chinese Academy of Sciences(CAS)

Agenda

- Review of existed hardware prefetchers
  - Questionable simulation prevails
  - No RTL-level implementation & evaluation
  - Industry interest is limited
- Why we choose RISC-V
  - Opensource, fairly good baseline
  - RISC-V implementation also lacks hardware prefetcher
  - Interests aligned
- Current Progress
  - Baseline Implementation
  - Adding L2 cache to RISC-V processor

Review

15 hardware prefetcher designs published on top4 conferences during the last 8 years

- None of it has provided RTL-level implementation or evaluation
- We can (only) confirm that BOP[7], SPP[8] and TPC[10] have impact on industry designs so far

[1] Linearizing Irregular Memory Accesses for Improved Correlated Prefetching. MICRO 2013.

[2] Sandbox Prefetching - Safe Run-Time Evaluation of Aggressive Prefetchers. HPCA 2014.

[3] B-Fetch - Branch Prediction Directed Prefetching for Chip-Multiprocessors. MICRO 2014.

[4] IMP - Indirect Memory Prefetcher. MICRO 2015.

[5] Self-contained, accurate precomputation prefetching. MICRO 2015.

[6] Efficiently Prefetching Complex Address Patterns. MICRO 2015.

[7] Best-Offset Hardware Prefetching. HPCA 2016.

[8] Path Confidence based Lookahead Prefetching. MICRO 2016.

[9] Domino Temporal Data Prefetcher. HPCA 2018

[10] Division of Labor - A More Effective Approach to Prefetching. ISCA 2018

[11] Translation-Triggered Prefetching. ASPLOS 2017.

[12] Kill the Program Counter - Reconstructing Program Behavior in the Processor Cache Hierarchy. ASPLOS 2017.

[13] PACMan - Prefetch-Aware Cache Management for High Performance Caching. MICRO 2011.

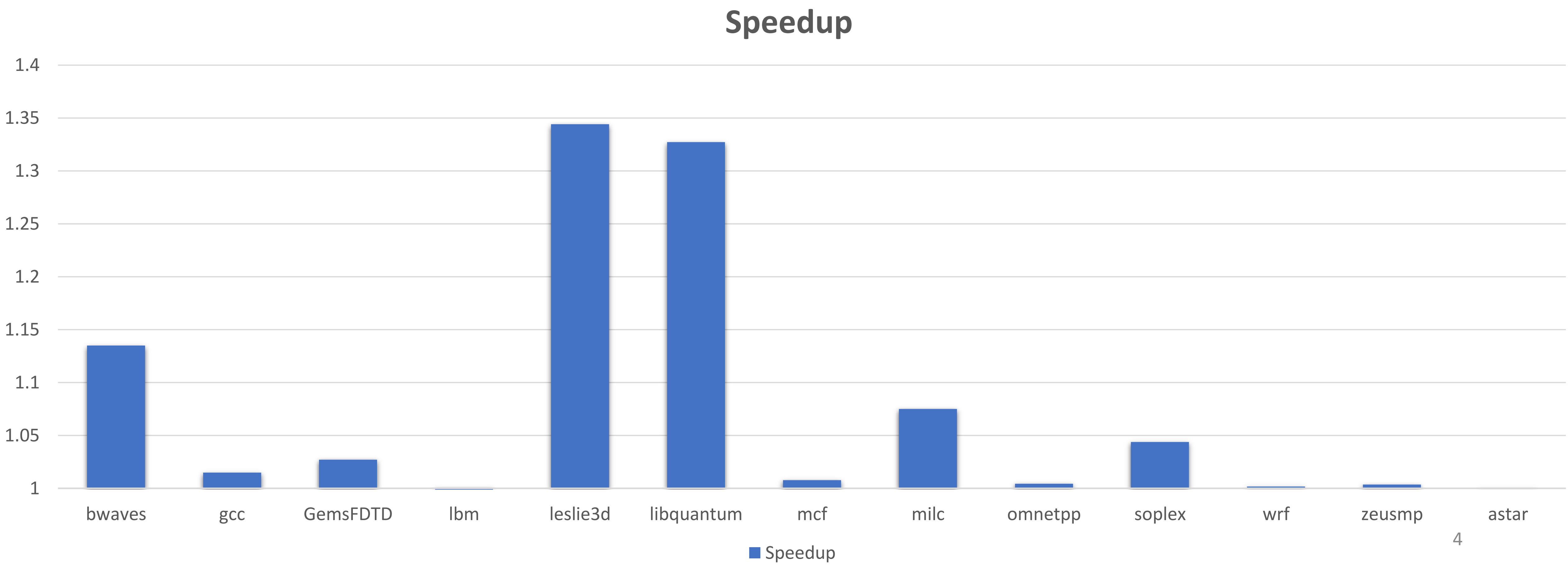
[14] Filtered Runahead Execution with a Runahead Buffer. MICRO 2015.

[15] Continuous Runahead - Transparent Hardware Acceleration for Memory Intensive Workloads. MICRO 2016.

- Review

Inaccurate simulation

- Industry has began to use exclusive cache design (Intel Skylake-EX / ARM Cortex-A55/A75)
- Non-inclusive cache *vs* Exclusive Cache



- Review

Inconsistent simulation

- **5% - 10%** speedup with **~100 cycle** DRAM latency
- **10% - 20%** speedup with **~290 cycle** DRAM latency
- Different DRAM model can reverse the performance rank of existed prefetchers

An example from calibrated simulator

	LLC MPKI	IPC	Useless Prefetch	DRAM read Latency
No LLC prefetcher	21.90	0.681	N/A	299 cycles
SPP	3.55	0.704	<0.01%	286 cycles
Testing Prefetcher	1.81	0.662	<0.01%	450 cycles

- MPKI can be utterly outweighed by the DRAM
- Our industry research partner confirmed this phenomenon

- Review

We need one step further … A new platform for architecture research

- Simulations are inaccurate
- Simulation results are inconsistent across papers
- Implementation difficulties are unexplored

Why RISC-V ?

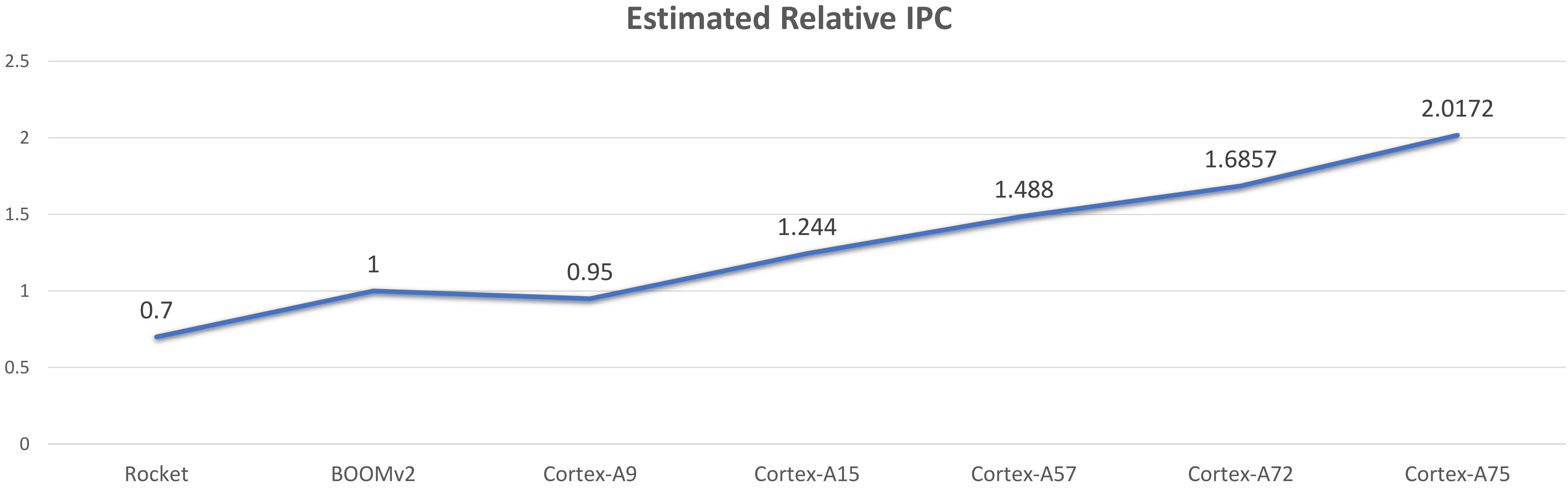
- Opensourced
- Linux-compatible
- Highest performance core we can find so far[1]
- We need RISC-V, and vice versa.

[1] OpenPiton: An Open Source Manycore Research Framework. ASPLOS 2016

Our interests are aligned

RISC-V needs to improve architectural performance

- Although BOOMv2[1] is reported to be faster than ARM Cortex A9 …
- Architectural performance bar of ARM's lineup[2][3] is rising **~25%** per generation



[1] BOOMv2 : an open-source out-of-order RISC-V core. RISC-V Workshop 2017

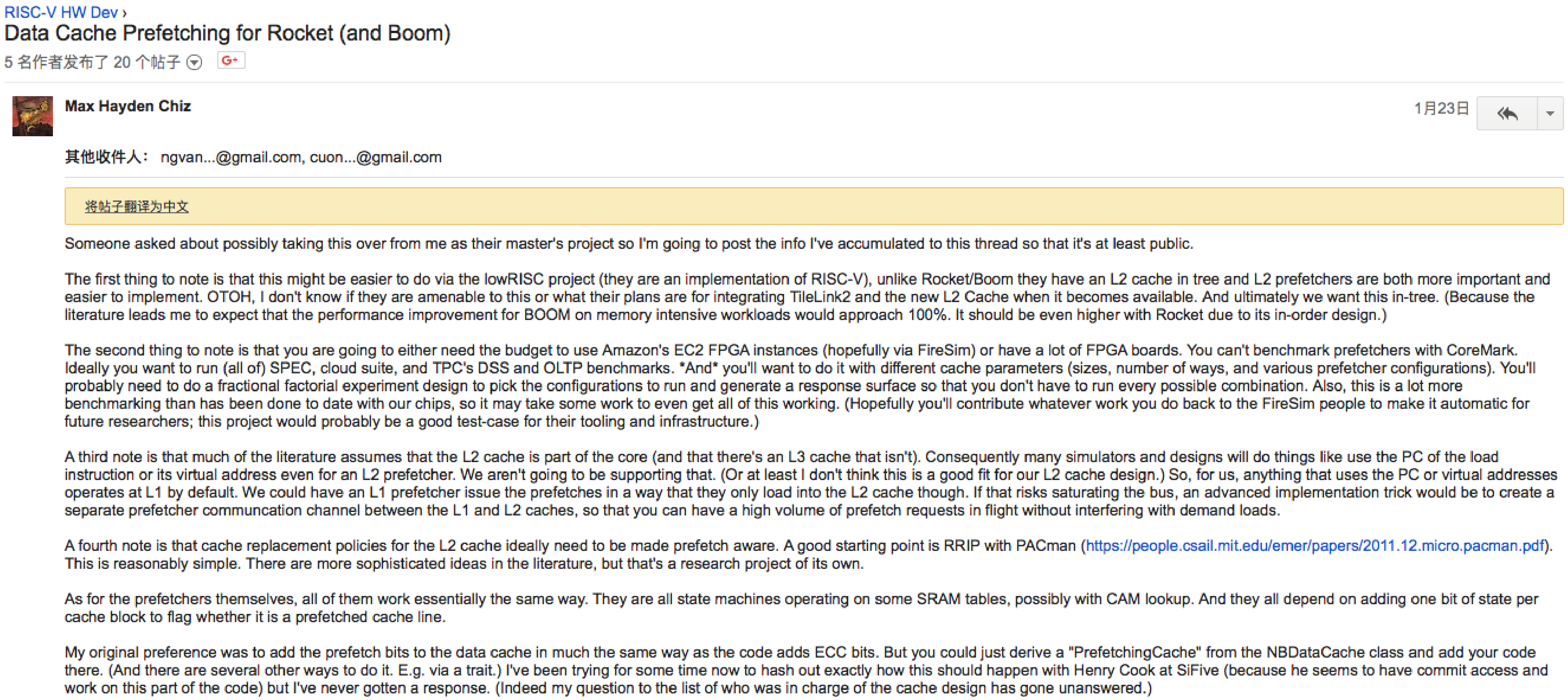
[2] The final ISA showdown: Is ARM, x86, or MIPS intrinsically more power efficient ? ExtremeTech 2014

[3] Exploring DynamIQ and ARM's New CPUs: Cortex-A75, Cortex-A55. AnandTech 2017.

Our interests are aligned

RISC-V community has already began to pay attention to prefetching

- Discussion can be found in “[hw-dev] Data Cache Prefetching for Rocket (and Boom)”



RISC-V HW Dev ›

Data Cache Prefetching for Rocket (and Boom)

5 名作者发布了 20 个帖子

Max Hayden Chiz

1月23日

其他收件人： ngvan...@gmail.com, cuon...@gmail.com

Someone asked about possibly taking this over from me as their master's project so I'm going to post the info I've accumulated to this thread so that it's at least public.

The first thing to note is that this might be easier to do via the lowRISC project (they are an implementation of RISC-V), unlike Rocket/Boom they have an L2 cache in tree and L2 prefetchers are both more important and easier to implement. OTOH, I don't know if they are amenable to this or what their plans are for integrating TileLink2 and the new L2 Cache when it becomes available. And ultimately we want this in-tree. (Because the literature leads me to expect that the performance improvement for BOOM on memory intensive workloads would approach 100%. It should be even higher with Rocket due to its in-order design.)

The second thing to note is that you are going to either need the budget to use Amazon's EC2 FPGA instances (hopefully via FireSim) or have a lot of FPGA boards. You can't benchmark prefetchers with CoreMark. Ideally you want to run (all of) SPEC, cloud suite, and TPC's DSS and OLTP benchmarks. *And* you'll want to do it with different cache parameters (sizes, number of ways, and various prefetcher configurations). You'll probably need to do a fractional factorial experiment design to pick the configurations to run and generate a response surface so that you don't have to run every possible combination. Also, this is a lot more benchmarking than has been done to date with our chips, so it may take some work to even get all of this working. (Hopefully you'll contribute whatever work you do back to the FireSim people to make it automatic for future researchers; this project would probably be a good test-case for their tooling and infrastructure.)

A third note is that much of the literature assumes that the L2 cache is part of the core (and that there's an L3 cache that isn't). Consequently many simulators and designs will do things like use the PC of the load instruction or its virtual address even for an L2 prefetcher. We aren't going to be supporting that. (Or at least I don't think this is a good fit for our L2 cache design.) So, for us, anything that uses the PC or virtual addresses operates at L1 by default. We could have an L1 prefetcher issue the prefetches in a way that they only load into the L2 cache though. If that risks saturating the bus, an advanced implementation trick would be to create a separate prefetcher communcation channel between the L1 and L2 caches, so that you can have a high volume of prefetch requests in flight without interfering with demand loads.

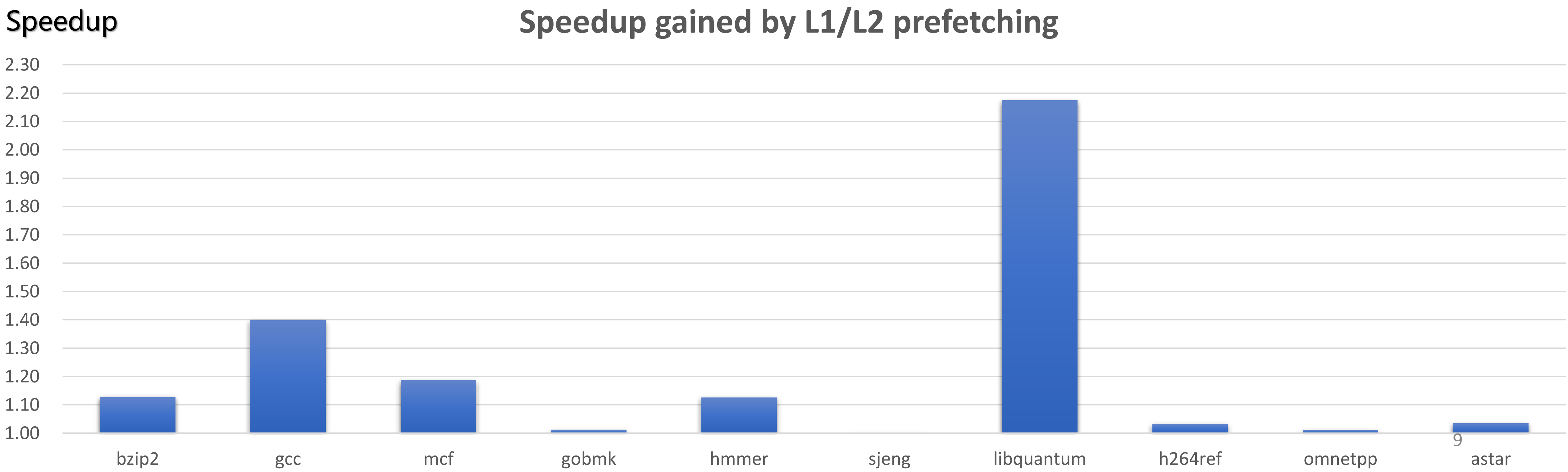
A fourth note is that cache replacement policies for the L2 cache ideally need to be made prefetch aware. A good starting point is RRIP with PACman (https://people.csail.mit.edu/emer/papers/2011.12.micro.pacman.pdf). This is reasonably simple. There are more sophisticated ideas in the literature, but that's a research project of its own.

As for the prefetchers themselves, all of them work essentially the same way. They are all state machines operating on some SRAM tables, possibly with CAM lookup. And they all depend on adding one bit of state per cache block to flag whether it is a prefetched cache line.

My original preference was to add the prefetch bits to the data cache in much the same way as the code adds ECC bits. But you could just derive a "PrefetchingCache" from the NBDataCache class and add your code there. (And there are several other ways to do it. E.g. via a trait.) I've been trying for some time now to hash out exactly how this should happen with Henry Cook at SiFive (because he seems to have commit access and work on this part of the code) but I've never gotten a response. (Indeed my question to the list of who was in charge of the cache design has gone unanswered.)

Prefetching Performance on Intel Haswell Xeon

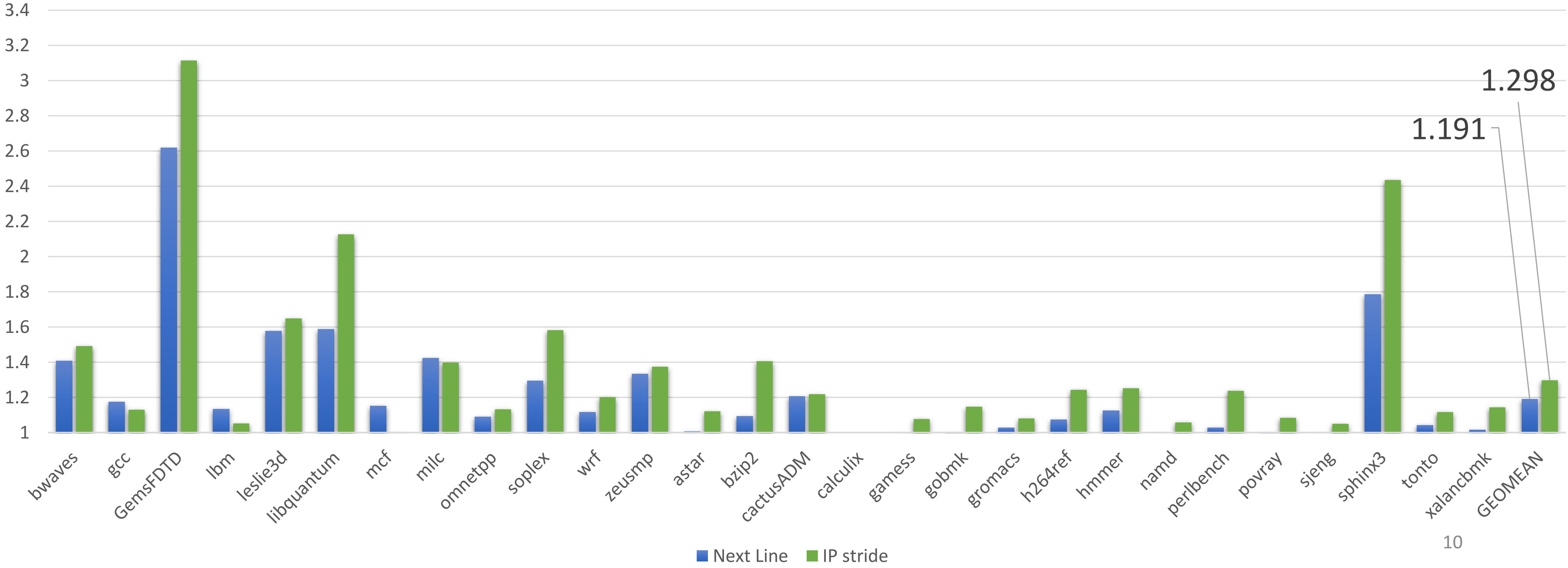
- Intel Xeon E5-2658v3 （Haswell, 2.2GHz, L3 Cache 30MB）
  - Running on a separated 3MB capacity of LLC by CAT, which is on par with current mobile SoC
  - gcc 4.8.5 -O3, linux kernel 3.10, 4KB page
- **20.8%** on average，**117%** for libquantum



L1 Prefetcher Prelim. Evaluation

Simulation result on a 4-issue BOOM model, tested with SPECCPU 2006

- Next Line Prefetcher ≈ +20% IPC
- Instruction-Pointer Based Stride Prefetcher ≈ **+30%** IPC



How can it be industry-competitive

Exploring more

- Connect prefetcher with TLB
  - **+13%** perf on milc *vs* **+60%** perf on milc (with paging info)
  - Our industry partners won't try this due to TLB bandwidth
- Hybrid Prefetcher
  - ARM hasn't adopted this, but we do know somebody implement this already

Exploring faster

- Real case from our industry partner
  - **90 man-months** to build a semi-new simulator
  - **A full day** to run **~1B** instructions
- Development with Chisel and FPGA emulation are much faster

- But …

The Official RISC-V implementation lacks L2 cache

- Prefetching can't be too aggressive, to avoid L1 cache pollution
- L1 cache MSHR and input/output buffer size are limited

Forked implementations are using outdated L2 cache

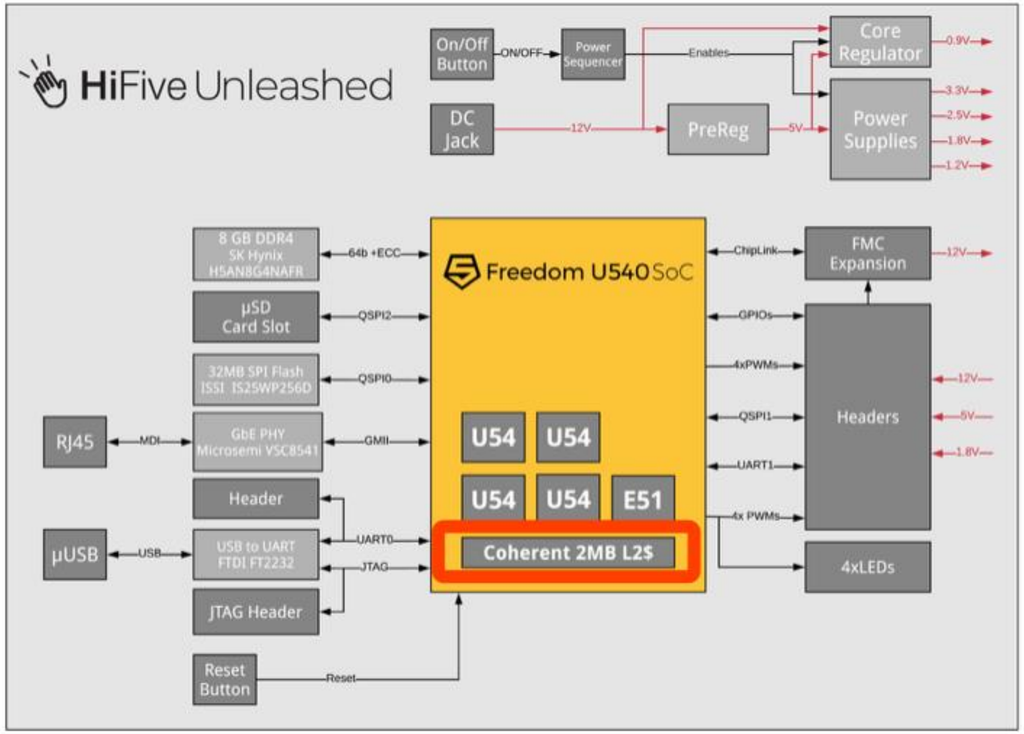
- Both LowRISC / Labeled RISC-V falls into this category

How can we solve that ？

Our interests are aligned

The official opensourced RISC-V implementation / SIFIVE products

- BOOM / Rocket both have parameterized L1 cache
- SIFIVE implement L2 cache for its commercial products[1]
  - 16-way set-associative
  - Multi-banked design
  - Scratchpad
  - Waymasking and locking
  - Cache-coherent
  - ECC



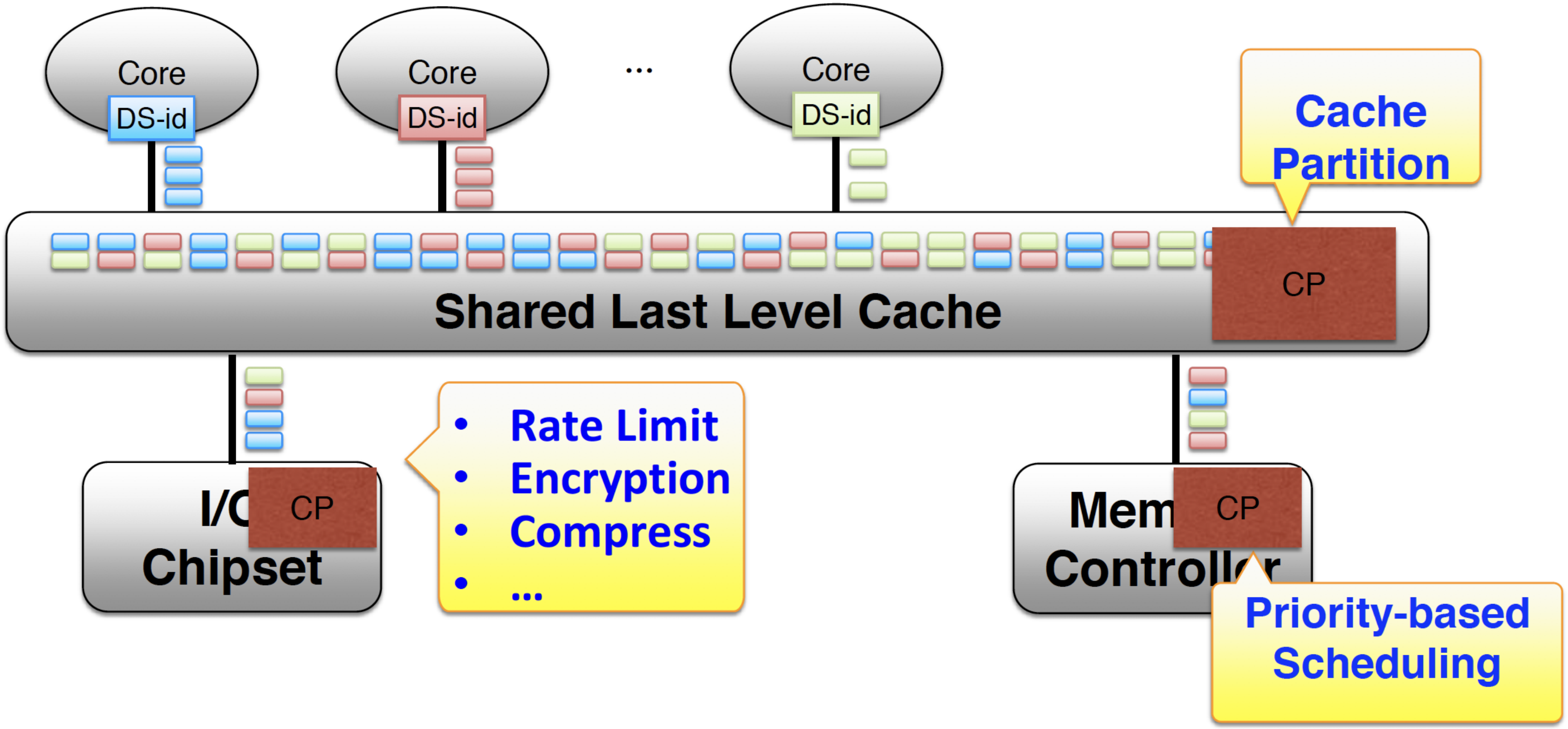
[1] https://www.sifive.com/products/hifive-unleashed/

[2] https://static.dev.sifive.com/FU540-C000-v1.0.pdf

Our interests are aligned

We (Labeled RISC-V[1] team) need high-performance L2 cache

- Our research heavily lies on shared cache & memory controller
  - Prefetch into L2 cache / Multi-layer prefetchers
  - (auto) cache capacity allocation, (auto) memory bandwidth allocation

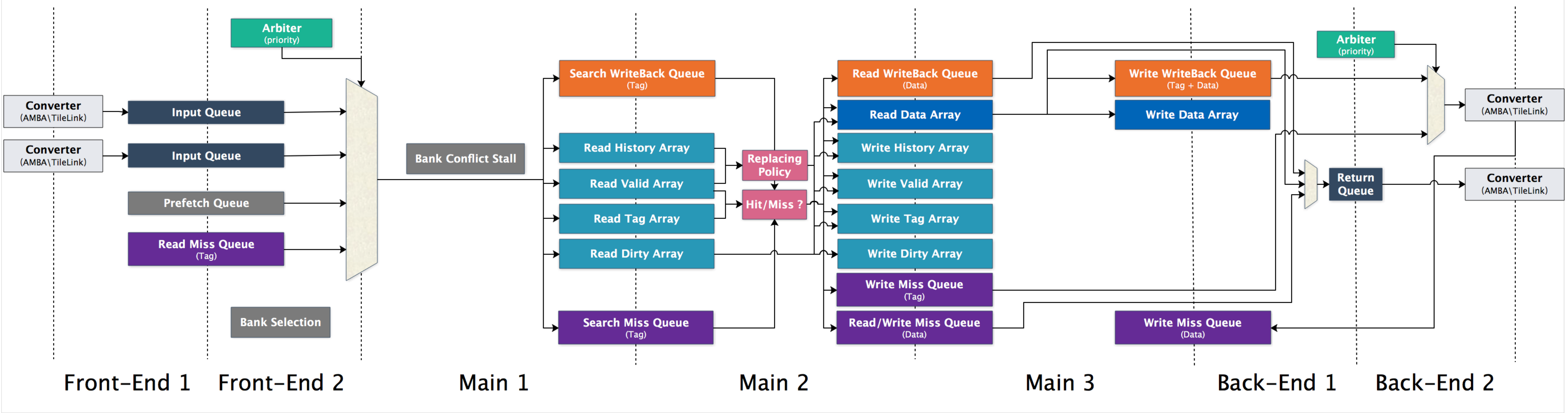


[1] Labeled RISC-V: A New Perspective on Software-Defined Architecture. CARRV 2017.

A New L2 Cache Design

Highlights

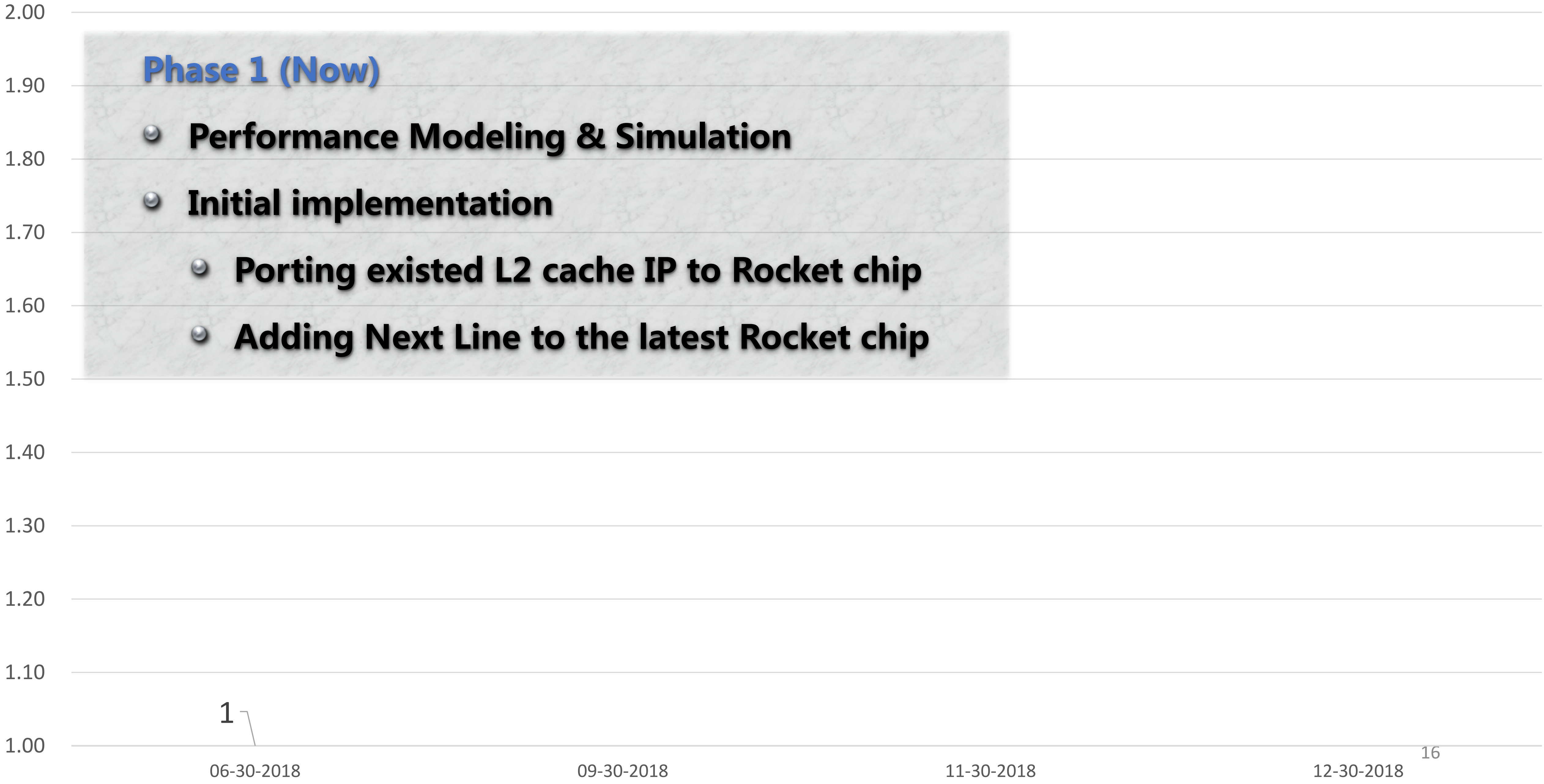
- Highly parameterized (capacity, associativity, buffer queue size, packet-based flow …)
- Heavy storage structures are implemented with single-port cell
- Multi-banked, 2-cycle same-bank-access stall
- Reserved port for future prefetcher implementation, 1-bit NRU replacing policy



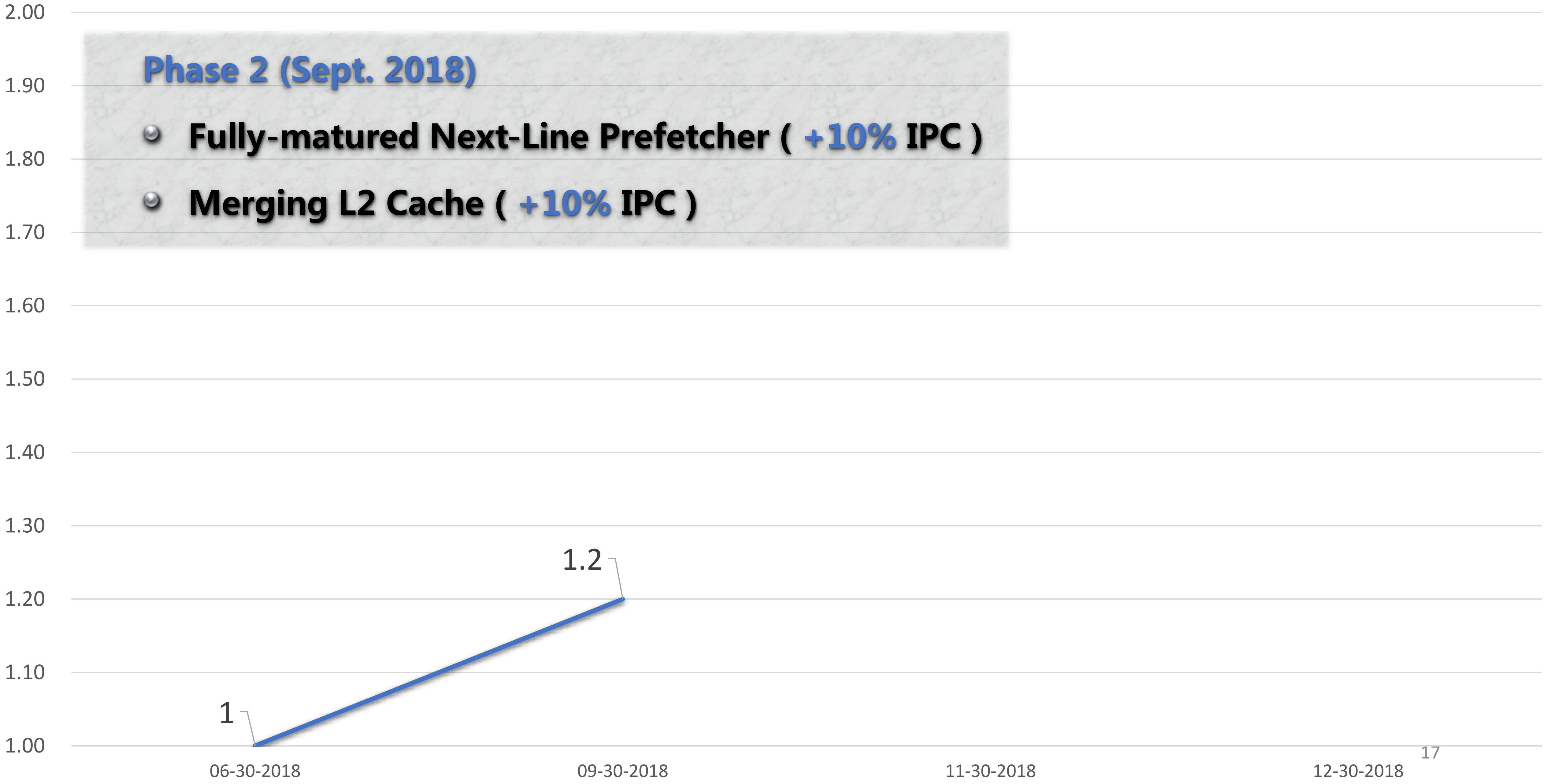
Relative Performance & Progress Projection

Phase 1 (Now)

- Performance Modeling & Simulation
- Initial implementation
  - Porting existed L2 cache IP to Rocket chip
  - Adding Next Line to the latest Rocket chip



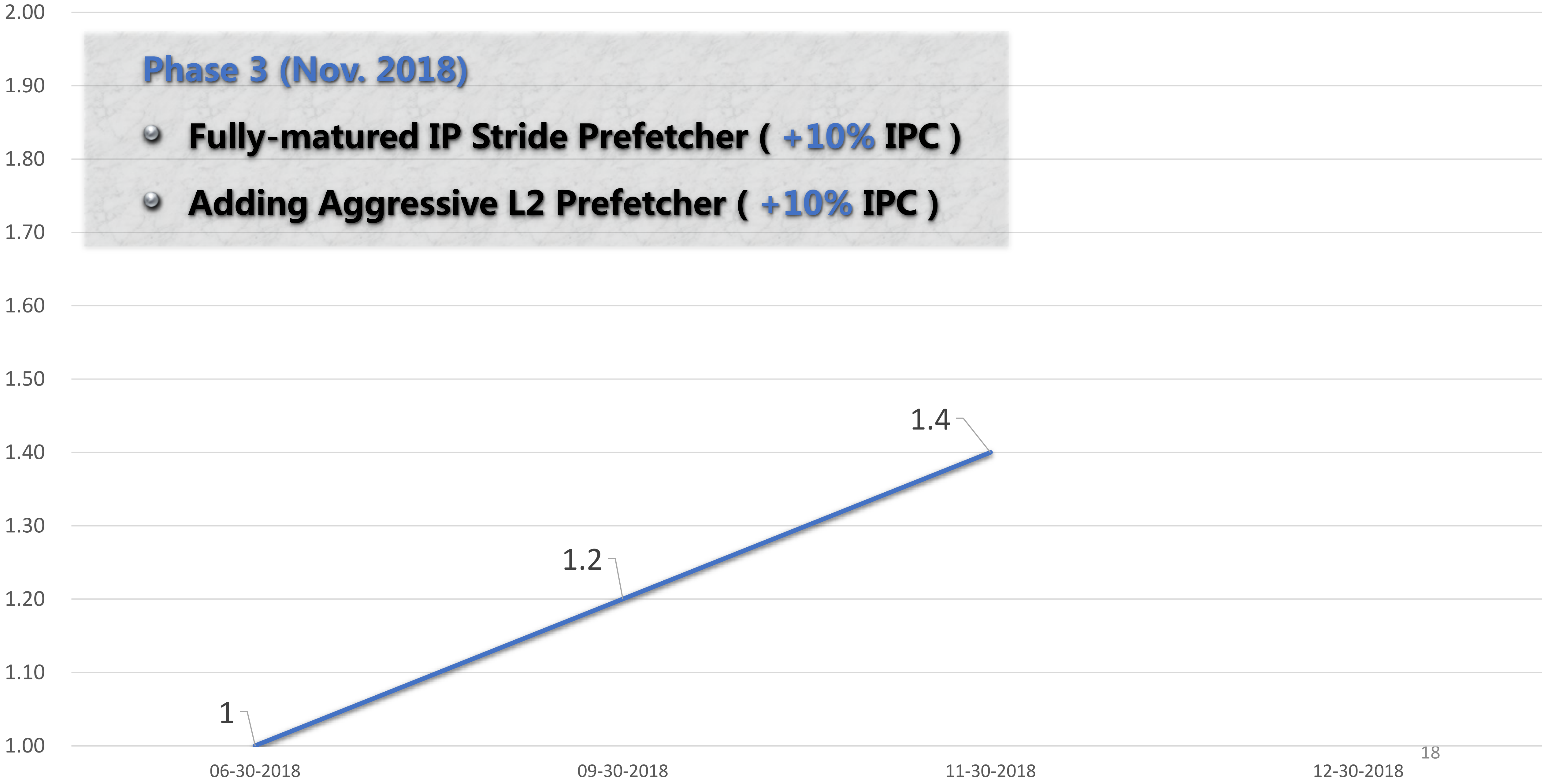
Relative Performance & Progress Projection



Phase 2 (Sept. 2018)

- **Fully-matured Next-Line Prefetcher (+10% IPC)**
- **Merging L2 Cache (+10% IPC)**

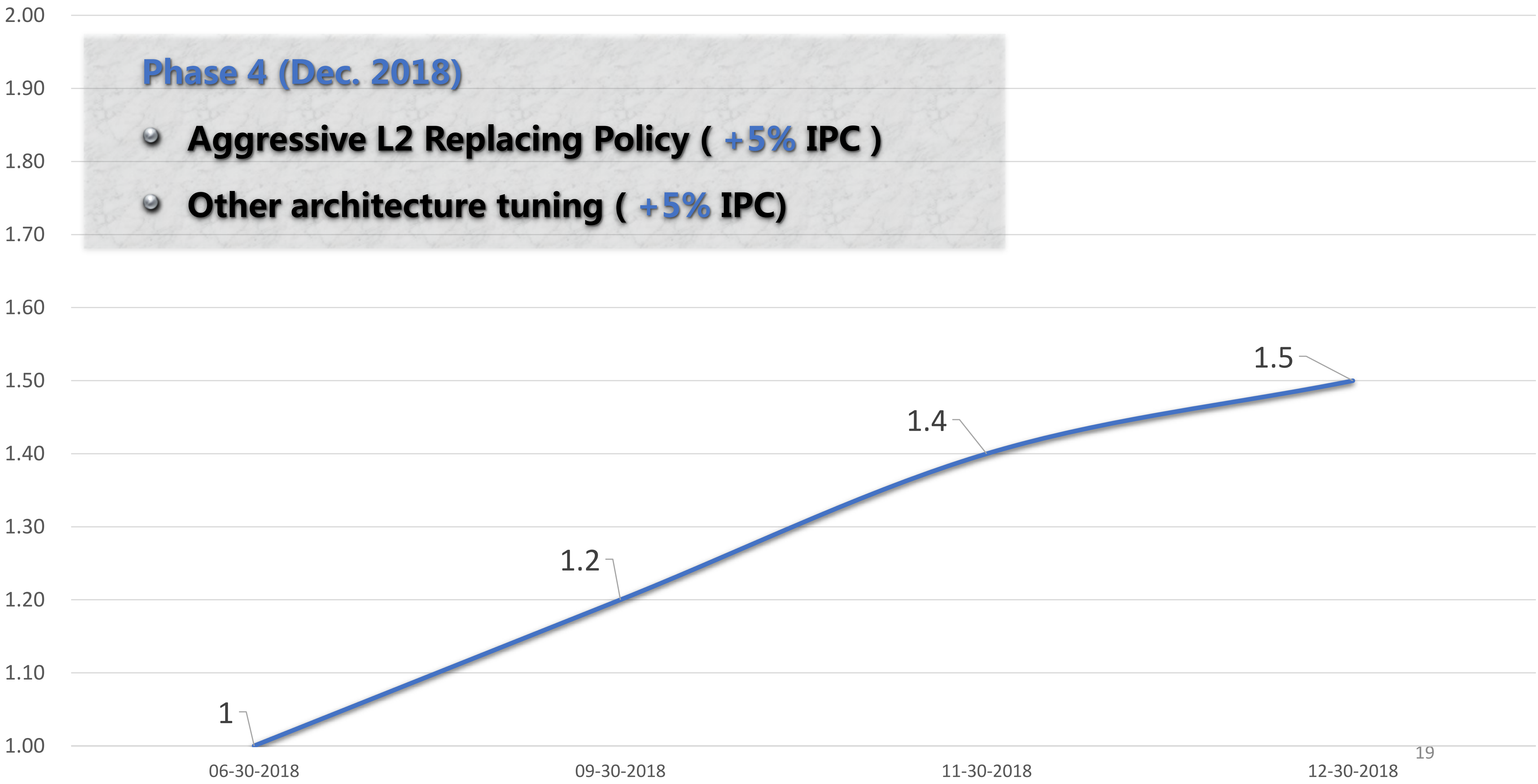
Relative Performance & Progress Projection



Phase 3 (Nov. 2018)

- **Fully-matured IP Stride Prefetcher (+10% IPC)**
- **Adding Aggressive L2 Prefetcher (+10% IPC)**

Relative Performance & Progress Projection



Phase 4 (Dec. 2018)

- **Aggressive L2 Replacing Policy (+5% IPC)**
- **Other architecture tuning (+5% IPC)**

Thanks !

- We can't make absolute promise, but will try hard to reach it
- Our implementation will be kept opensourced with Labeled RISC-V branch

  https://github.com/LvNA-system/labeled-RISC-V
- Also welcome volunteers
  - Testing more benchmarks
  - Merging with master branch or checkout a new dedicated branch