Evolution of the Samsung Exynos CPU Microarchitecture

Industrial Product

Brian Grayson*, Jeff Rupley†, Gerald Zuraski Jr.‡, Eric Quinnell§, Daniel A. Jiménez¶, Tarun Nakra‡‡, Paul Kitchin**, Ryan Hensley††, Edward Brekelbaum*, Vikas Sinha**, and Ankit Ghiya§

bgrayson@ieee.org, jrupley@austin.rr.com, gzuraskijr@gmail.com, eric.quinnell@gmail.com, djimenez@acm.org, Tarun.Nakra@amd.com, pkitchin@gmail.com, ryan.hensley@ieee.org, nedbrek@gmail.com, sinhavk@gmail.com, and mascot26@gmail.com

*Sifive †Centaur ‡Independent Consultant §ARM ¶Texas A&M University ‡‡AMD **Nuvia ††Goodix

***Abstract*—The Samsung Exynos family of cores are high-performance "big" processors developed at the Samsung Austin Research & Design Center (SARC) starting in late 2011. This paper discusses selected aspects of the microarchitecture of these cores - specifically perceptron-based branch prediction, Spectre v2 security enhancements, micro-operation cache algorithms, prefetcher advancements, and memory latency optimizations. Each micro-architecture item evolved over time, both as part of continuous yearly improvement, and in reaction to changing mobile workloads.**

***Index Terms*—microprocessor, superscalar, branch prediction, prefetching**

I. Introduction

Samsung started development of the Exynos family of cores beginning in late 2011, with the first generation core ("M1") supporting the newly introduced ARM v8 64-bit architecture [1]. Several generations of the Exynos M-series CPUs [2] [3], here referred to as M1 through M6, are found most commonly in the Samsung Exynos-based Galaxy S7 through S20 smart phones, and are implemented in a variety of process nodes, from 14nm to 7nm. The cores support the ARMv8 Instruction Set Architecture [4], both AArch32 and AArch64 variants. The cores are superscalar out-of-order designs capable of up to 2.9GHz, and employ up to three levels of cache hierarchy in a multi-node cluster. Each Exynos M-series CPU cluster is complemented by an ARM Cortex-A series cluster in a big/little configuration in generations one through three, and in a big/medium/little configuration in subsequent generations.

Among the numerous details from this effort, this paper selectively covers:

- Yearly evolution of a productized core microarchitecture (M1 through M5) and future work (M6);
- Adaptations of the microarchitecture due to changes in mobile workloads;
- Deep technical details within the microarchitecture.

The remainder of this paper discusses several aspects of front-end microarchitecture (including branch prediction microarchitecture, security mitigations, and instruction supply) as well as details on the memory subsystem, in particular with regards to prefetching and DRAM latency optimization. The overall generational impact of these and other changes is presented in a cross-workload view of IPC.

This paper is part of the Industry Track of ISCA 2020's program. All of the authors were employed by Samsung at SARC while working on the cores described here. The authors express gratitude to Samsung for supporting the publication of this paper.

II. Methodology

Simulation results shown here as well as the internal microarchitectural tuning work are based on a trace-driven cycle-accurate performance model that reflects all six of the implementations in this paper, rather than a silicon comparison, because M5 and M6 silicon were not available during the creation of this paper.

The workload is comprised of 4,026 traces, gathered from multiple CPU-based suites such as SPEC CPU2000 and SPEC CPU2006; web suites including Speedometer, Octane, BBench, and SunSpider; mobile suites such as AnTuTu and Geekbench; and popular mobile games and applications. SimPoint [5] and related techniques are used to reduce the simulation run time for most workloads, with a warmup of 10M instructions and a detailed simulation of the subsequent 100M instructions. Note that the same set of workloads is utilized here across all of the generations; some of the more recent workloads did not exist during the development of M1, and some earlier workloads may have become somewhat obsolete by M6, but keeping the workload suite constant allows a fair cross-generational comparison.

III. Microarchitectural Overview

Several of the key microarchitectural features of the M1 through M6 cores are shown in Table I. Although the cores were productized at different frequencies, performance results in this paper come from simulations where all cores were run at 2.6GHz, so that per-cycle comparisons (IPC, load latencies)

are valid to compare. Selected portions of the microarchitecture that are not covered elsewhere will be briefly discussed in this section.

For data translation operations, M3 and later cores contain a fast "level 1.5 Data TLB" to provide additional capacity at much lower latency than the much-larger L2 TLB.

Note that, according to the table, there were no significant resource changes from M1 to M2. However, there were several efficiency improvements, including a number of deeper queues not shown in Table I, that resulted in the M2 speedups shown later in this paper.

Both the integer and floating-point register files utilize the physical-register-file (PRF) approach for register renaming, and M3 and newer cores implement zero-cycle integer register-register moves via rename remapping and reference-counting.

The M4 core and beyond have a "load-load cascading" feature, where a load can forward its result to a subsequent load a cycle earlier than usual, giving the first load an effective latency of 3 cycles.

In general, most resources are increased in size in succeeding generations, but there are also several points where sizes were reduced, due to evolving tradeoffs. Two examples are M3's reduction in L2 size due to the change from shared to private L2 as well as the addition of an L3, and M4's reduction in L3 size due to changing from a 4-core cluster to a 2-core cluster.

IV. Branch Prediction

The Samsung dynamic branch prediction research is rooted in the Scaled Hashed Perceptron (SHP) approach [6] [7] [8] [9] [10], advancing the state-of-the-art perceptron predictor over multiple generations. The prediction hardware uses a Branch Target Buffer (BTB) approach across both a smaller, 0-bubble TAKEN micro-BTB (μBTB) with a local-history hashed perceptron (LHP) and a larger 1-2 bubble TAKEN main-BTB (mBTB) with a full SHP. The hardware retains learned information in a Level-2 BTB (L2BTB) and has a virtual address-based BTB (vBTB) in cases of dense branch lines that spill the normal BTB capacity. Function returns are predicted with a Return-Address Stack (RAS) with standard mechanisms to repair multiple speculative pushes and pops.

A. Initial direction

The initial branch predictor design had two performance goals: use a state-of the-art conditional predictor to reduce MPKI, and support up to two predictions per clock for the common case of a leading NOT-TAKEN branch. The latter avoids incurring an extra cycle (or more) for many of the NOT-TAKEN branches. For the workloads discussed here, the lead branch is TAKEN in 60% of all cases, the second paired branch TAKEN 24% of all cases, and two sequential NOT-TAKEN 16% of all cases.

The first generation SHP consists of eight tables of 1,024 weights each, in sign/magnitude representation, along with a "local BIAS" weight kept in the BTB entry for each branch. Each of the SHP weight tables is indexed using an XOR hash [11] composed of three items:

1) A hash of the global history (GHIST) [12] pattern in a given interval for that table. The GHIST records the outcome of a conditional branch as one bit.
2) A hash of the path history (PHIST) [13] in a given interval for that table. The PHIST records three bits, bits two through four, of each branch address encountered.
3) A hash of the program counter (PC) for the branch being predicted.

The GHIST and PHIST intervals were determined empirically using a stochastic search algorithm, taking into consideration both the diminishing returns of longer GHIST (shown in Figure 1 using the publicly-available CBP5 [14] workloads), and the cost of maintaining GHIST state. M1 utilized a 165-bit GHIST length, and an 80-bit PHIST length.

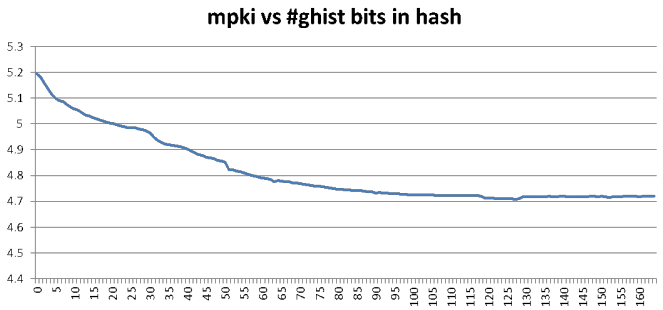

Fig. 1. Average MPKI over all CBP5 traces of an eight-table, 1K weight SHP vs adding GHIST range bits to the hash

To compute an SHP prediction, the signed BIAS weight in the BTB entry is doubled [8], and added to the sum of the signed weights read out from each of the 8 tables according to the hashed history intervals. If the resulting sum is at least 0, then the branch is predicted TAKEN; otherwise, it is predicted not taken.

The predictor is updated on a misprediction, or on a correct prediction where the absolute value of the sum fails to exceed a threshold trained using the threshold training algorithm from the O-GEHL predictor [15]. To update the SHP predictor, the counters used during prediction are incremented if the branch was taken, or decremented otherwise, saturating at the maximum or minimum counter values. Always-TAKEN branches — both unconditional branches, and conditional branches that have always been taken so far — do not update the SHP weight tables, to reduce the impact of aliasing [16].

The main BTBs are organized into 8 sequential discovered branches per 128B cacheline as seen in Figure 2, based on the gross average of 5 instructions per branch. As mentioned above, additional dense branches exceeding the first 8 spill to a virtual-indexed vBTB shown in the right of Figure 2 at an additional access latency cost.

The indirect branch predictor is based on the virtual program counter (VPC) predictor [17]. VPC breaks up an indirect prediction into a sequence of conditional predictions of "virtual

TABLE I
MICROARCHITECTURAL FEATURE COMPARISON

	Core name					
Feature	M1	M2	M3	M4	M5	M6
Process Node	14nm	10nm LPE	10nm LPP	8nm LPP	7nm	5nm
Product Frequency	2.6GHz	2.3GHz	2.7GHz	2.7GHz	2.8GHz	2.8GHz (target)
Caches						
L1 Instruction Cache	64KB 4w	64KB 4w	64KB 4w	64KB 4w	64KB 4w	128KB 4w
L1 Data Cache	32KB 8w	32KB 8w	64KB 8w	64KB 4w	64KB 4w	128KB 8w
L2 Cache	2048KB 16w	2048KB 16w	512KB 8w	1024KB 8w	2048KB 8w	2048KB 8w
L2 Shared/Private	shared by 4 cores	shared by 4 cores	private	private	shared by 2 cores	shared by 2 cores
L2 BW	16B/cycle	16B/cycle	32B/cycle	32B/cycle	32B/cycle	64B/cycle
L3 Cache Size	-	-	4096KB	3072KB	3072KB	4096KB
L3 Ways & Banks	-	-	16w 4 bank	16w 3 bank	12w 2 bank	16w 2 bank
Translation[a]						
L1 Instruction TLB	256 pgs (64/64/4)	256 pgs (64/64/4)	512 pgs (64/64/8)	512 pgs (64/64/8)	512 pgs (64/64/8)	512 pgs (64/64/8)
L1 Data TLB	32 pgs (32/32/1)	32 pgs (32/32/1)	32 pgs (32/32/1)	48 pgs (48/48/1)	48 pgs (48/48/1)	128 pgs (128/128/1)
L1.5 Data TLB	-	-	512 pgs (128/4/4)	512 pgs (128/4/4)	512 pgs (128/4/4)	512 pgs (128/4/4)
Shared L2 TLB	1K pgs (1K/4/1)	1K pgs (1K/4/1)	4K pgs (1K/4/4)	4K pgs (1K/4/4)	4K pgs (1K/4/4)	8K pgs (2K/4/4)
Execution Unit Details						
Dec/Ren/Ret width	4	4	6	6	6	8
Integer units[b]	2S+1CD+BR	2S+1CD+BR	2S+1CD+1C+BR	2S+1CD+1C+BR	4S+1CD+1C+BR	4S+2CD+2BR
Ld/St/Generic[c] pipes	1L, 1S	1L, 1S	2L, 1S	1L, 1S, 1G	1L, 1S, 1G	1L, 1S, 1G
FP pipes	1FMAC, 1FADD	1FMAC, 1FADD	3FMAC	3FMAC	3FMAC	4FMAC
Integer PRFs	96	96	192	192	192	224
FP PRFs	96	96	192	176	176	224
ROB size	96	100	228	228	228	256
Latencies						
Mispredict penalty	14	14	16	16	16	16
L1 hit latency	4	4	4	3[d] or 4	3[d] or 4	3[d] or 4
L2 avg. latency	22	22	12	12	13.5	13.5
L3 avg. latency	-	-	37	37	30	30
FP latencies[e]	5/4/3	5/4/3	4/3/2	4/3/2	4/3/2	4/3/2

[a]Translation parameters are shown as *total pages (#entries / #ways / #sectors)*
[b]"S ALUs handle add/shift/logical; C ALUs handle simple plus mul/indirect-branch; CD ALUs handle C plus div; BR handle only direct branches
[c]"Generic" units can perform either loads or stores
[d]Load-to-load cascading has a latency of only 3 cycles
[e]FP latencies are shown in cycles for FMAC/FMUL/FADD respectively

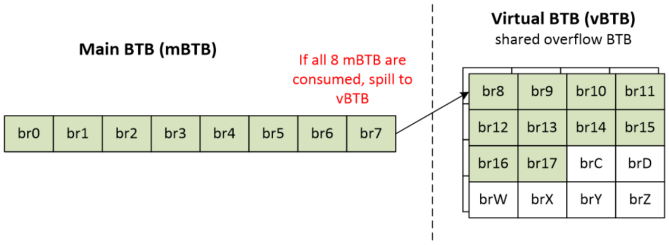

Fig. 2. Main and Virtual BTB branch "chains"

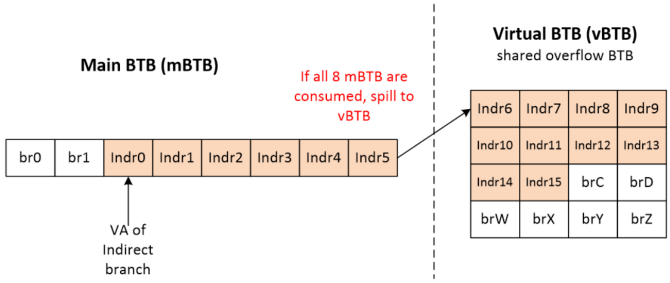

Fig. 3. Main and Virtual BTB indirect VPC chain

PCs" that each consult SHP, with each unique target up to a design-specified maximum "chain" stored in the BTB at the program order of the indirect branch. Figure 3 shows the VPC algorithm with a maximum of 16 targets per indirect branch, several of which are stored in the shared vBTB.

B. First refinement during M1 design

During early discussions, the two-bubble penalty on TAKEN branches was clearly identified as limiting in certain scenarios, such as tight loops or code with small basic blocks and predictable branches. The baseline predictor is therefore augmented with a micro-BTB (μBTB) that has zero-bubble throughput, but limited capacity. This predictor is graph-based [18] specifically using an algorithm to first filter and identify common branches with common roots or "seeds" and then learn both TAKEN and NOT-TAKEN edges into a "graph" across several iterations, as seen in the example in Figure 4. Difficult-to-predict branch nodes are augmented with use of a

local-history hashed perceptron (LHP).

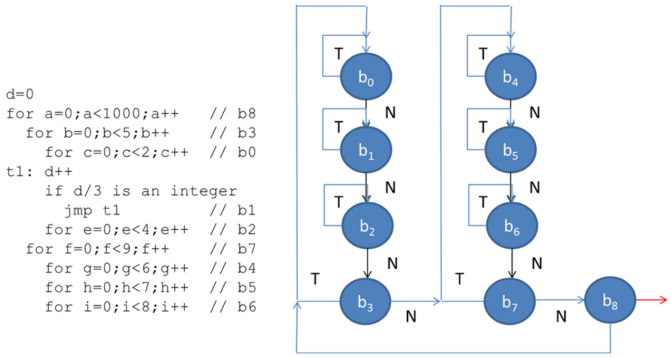

Fig. 4. Learned branch "graph" as used by the µBTB [18]

When a small kernel is confirmed as both fully fitting within the µBTB and predictable by the µBTB, the µBTB will "lock" and drive the pipe at 0 bubble throughput until a misprediction, with predictions checked by the mBTB and SHP. Extremely highly confident predictions will further clock gate the mBTB for large power savings, disabling the SHP completely.

The above description completes an overview of the branch prediction hardware in the M1 first-generation core. The M2 core made no significant changes to branch prediction.

C. M3 Branch Prediction: Throughput

For the M3 implementation, the rest of the core was undergoing significant widening of the pipe (4-wide to 6-wide throughout) as well as more than doubling of the out-of-order window. To maintain the ability to feed a wider and more capable microarchitecture, the branch predictor needed to be improved.

To reduce average branch turnaround, the µBTB graph size doubled, but reduced the area increase by only adding entries that could be used exclusively to store unconditional branches. For workloads that could not fit within the µBTB, the concept of an early always-taken redirect was added for the mBTB: Always-taken branches will redirect a cycle earlier. Such branches are called 1AT branches, short for "1-bubble always-TAKEN" branches.

M3 also made several changes to further reduce MPKI in the predictor: doubling of SHP rows (reduces aliasing for conditional branches); and doubling of L2BTB capacity.

D. M4 Branch Prediction Changes: Focus on larger workloads

The M4 generation predictor refinement focused on better support for larger working set sizes. The L2BTB size was doubled again, making this generation capable of holding four times as many branches as the first generation. In addition, L2BTB fills into the BTB had their latency slightly reduced, and their bandwidth improved by 2x, emphasizing the importance of real-use-case code. This capacity and latency bandwidth change significantly improved performance for mobile workloads like BBench by 2.8% in isolation.

E. M5 Branch Prediction: Efficiency improvements

For the M5 generation, the branch predictor research focus was on improving the predictor in terms of several different efficiencies.

Some sections of code are very branch-sparse, sometimes with entire BTB lines devoid of any actual discovered branches. Thus the design added state to track "Empty Line Optimizations" (skip an entire line devoid of branches) to reduce both the latency and power of looking up uninteresting addresses.

The previous generation had already accelerated always-TAKEN branches via a one-bubble redirect. In this generation, this idea was extended in two different ways: reducing the bubbles to zero via replication, and extending coverage to often-TAKEN branches. The replication is done via copying the targets of always -TAKEN and often-TAKEN branches into their predecessor branches' mBTB information, thus providing zero-bubble always-TAKEN (ZAT) and zero-bubble often-TAKEN (ZOT) prediction capability. Thus, on an mBTB lookup for a branch, it provided both its own target, and if that target location led next to a candidate always/often-TAKEN branch, the target of that subsequent branch. This increased storage cost, but greatly improved taken-branch throughput, as shown in the example in Figure 5.

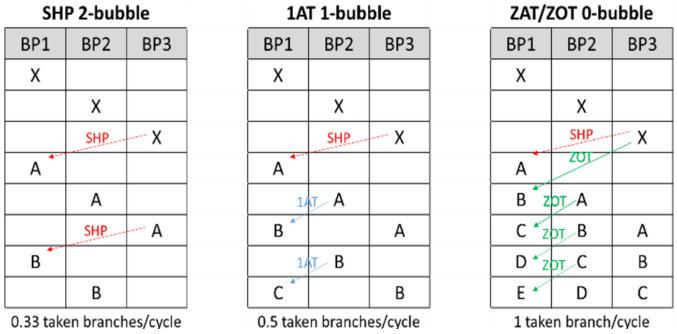

Fig. 5. Increase in branch throughput by reducing bubbles for Always Taken (AT) and zero-often-taken (ZOT) predictions. The replication scheme allows X to specify a redirect to both A, and to B, the target of the next branch after address A.

With a new zero-bubble structure in the mBTB, the µBTB and this ZAT/ZOT predictor can sometimes contend. The main predictor has no startup penalty but uses full mBTB and SHP resources, while the µBTB has a two-cycle startup penalty, but can chain zero-bubble predictions without lead branches and could also save mBTB and SHP power on tight kernels. A heuristic arbiter is used to intelligently select which zero-bubble predictor to use.

Given the growing capabilities of the one-bubble predictor, the M5 design was able to decrease the area for the µBTB by reducing the number of entries, and having the ZAT/ZOT predictor participate more. This resulted in a better area efficiency for a given amount of performance.

The SHP increased from 8 tables to 16 tables of 2,048 8-bit weights in sign magnitude representation, reducing MPKI from both alias reductions as well as increasing the number of

unique tables. The GHIST length was also increased by 25%, and the table hash intervals rebalanced for this longer GHIST.

Finally, one performance problem was the effective refill time after a mispredict. If the correct-path contained several small basic blocks connected with taken branches, it could take many cycles to refill the pipe, which was observed in some of our key web workloads, among others. This is shown in Figure 6, where we assume basic block A has 5 instructions, basic block B has 6, and basic block C has 3, and all end with a taken branch. Once a mispredict is signaled, the branch prediction pipe starts fetching at the correct target, A, and in subsequent cycles predicts sequential fall-through with a fetch width of 6, at addresses A+6 and A+12. (After a mispredict, the μBTB is disabled until the next "seed" branch.) However, when the branch prediction pipe reaches the third stage, it sees that A has a predicted-taken branch to B. The core squashes the speculative prediction lookups for A+6 and A+12, and restarts the branch prediction pipe for address B. In this example, assume A's basic block contained 5 instructions. If B also has a small basic block and a taken branch, it again requires three cycles to fetch an additional 6 instructions. All told, in this example, it requires 9 cycles from the mispredict redirect to fetch 14 instructions, and the downstream core will likely become fetch-starved.

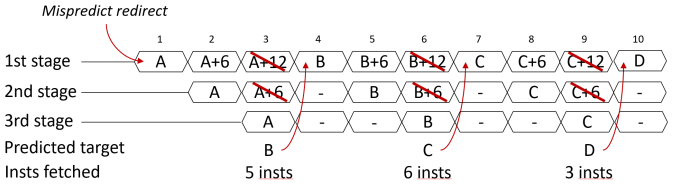

Fig. 6. An illustration of the slow effective refill time after a mispredict to a series of small basic blocks

In order to help mitigate this long effective refill time after a mispredict, the M5 core added a block called the Mispredict Recovery Buffer (MRB). After identifying low-confidence branches [19], it records the highest probability sequence of the next three fetch addresses. On a matching mispredict redirect, the predicted fetch addresses can be read sequentially out of the MRB in consecutive cycles as shown in Figure 7, eliminating the usual branch prediction delay, and in this example allowing 14 instructions to be provided in only 5 cycles. Note that in the third stage, the MRB-predicted target address is checked against the newly predicted target from the branch predictor, and if they agree, no correction is needed.

F. M6 Branch Prediction: Indirect capacity improvements

For the sixth generation, the greatest adjustment was to respond to changes in popular languages and programming styles in the market. The first of these changes was to increase the size of the mBTB by 50%, due to larger working set sizes.

Second, JavaScript's increased use put more pressure on indirect targets, allocating in some cases hundreds of unique indirect targets for a given indirect branch via function calls. Unfortunately, the VPC algorithm requires $O(n)$ cycles to train

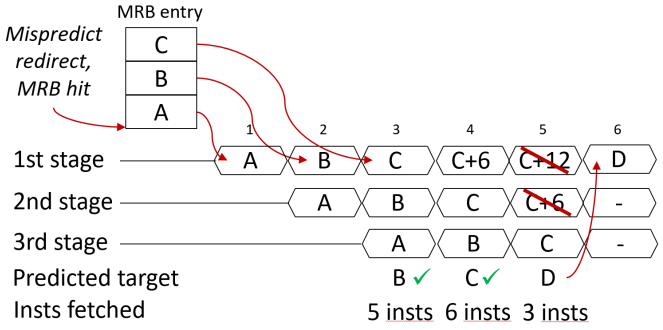

Fig. 7. Mispredict Recovery Buffer (MRB) providing the highest probability basic block addresses after an identified low-confidence mispredicted branch [20]

and predict all the virtual branches for an n-target indirect branch. It also consumes much of the vBTB for such many-target indirect branches.

A solution to the large target count problem is to dedicate unique storage for branches with very large target possibilities. As a trade-off, the VPC algorithm is retained since the accuracy of SHP+VPC+hash-table lookups still proves superior to a pure hash-table lookup for small numbers of targets. Additionally, large dedicated storage takes a few cycles to access, so doing a limited-length VPC in parallel with the launch of the hash-table lookup proved to be superior in throughput and performance. This hybrid approach is shown in Figure 8, and reduced end-to-end prediction latency compared to the full-VPC approach.

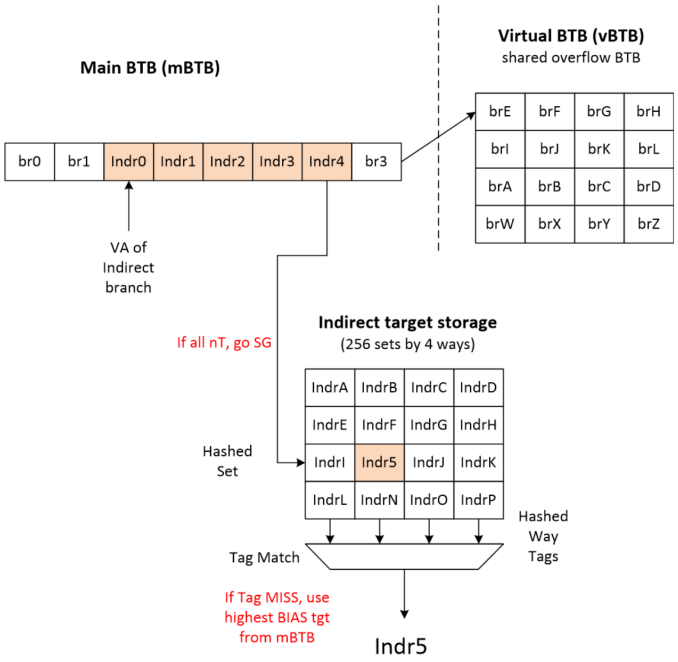

Fig. 8. VPC reduction to 5 targets followed by an Indirect Hash

Performance modeling showed that accessing the hash-table lookup indirect target table with the standard SHP

GHIST/PHIST/PC hash did not perform well, as the precursor conditional branches do not highly correlate with the indirect targets. A different hash is used for this table, based on the history of recent indirect branch targets.

G. Overall impact

From M1 through M6, the total bit budget for branch prediction increased greatly, in part due to the challenges of predicting web workloads with their much larger working set. Table II shows the total bit budget for the SHP, L1, and L2 branch predictor structures. The L2BTB uses a slower denser macro as part of a latency/area tradeoff.

TABLE II
BRANCH PREDICTOR STORAGE, IN KBYTES

Bit storage (KB)	SHP	L1BTBs	L2BTB	Total
M1/M2	8.0	32.5	58.4	98.9
M3	16.0	49.0	110.8	175.8
M4	16.0	50.5	221.5	288.0
M5	32.0	53.3	225.5	310.8
M6	32.0	78.5	451.0	561.5

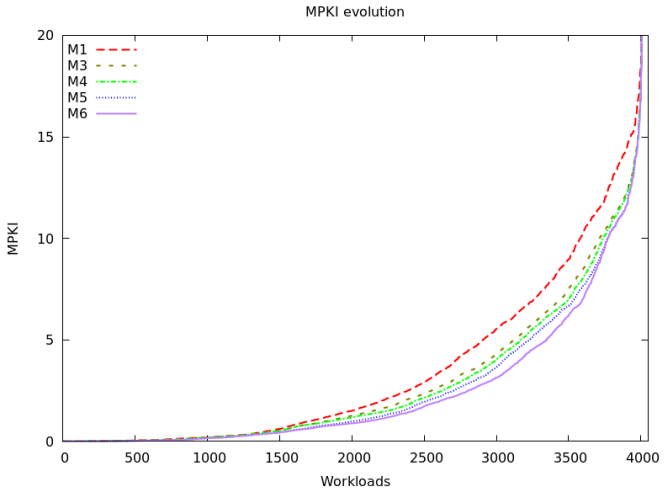

Fig. 9. MPKIs across 4,026 workload slices. Y-axis is clipped at 20 MPKI for clarity. Note that M2, which had no substantial branch prediction change over M1, is not shown in this graph.

With all of the above changes, the predictor was able to go from an average mispredicts-per-thousand-instructions (MPKI) of 3.62 for a set of several thousand workload slices on the first implementation, to an MPKI of 2.54 for the latest implementation. This is shown graphically in Figure 9, where the breadth of the impact can be seen more clearly.

On the left side of the graph, many workloads are highly predictable, and are unaffected by further improvements. In the middle of the graph are the interesting workloads, like most of SPECint and Geekbench, where better predictor schemes and more resources have a noticeable impact on reducing MPKI. On the right are the workloads with very hard to predict branches The graph has the Y-axis clipped to highlight the bulk of the workloads which have the characteristic of an MPKI under 20, but even the clipped highly-unpredictable workloads on M1 are improved by ~20% on subsequent generations. Overall, these changes reduced SPECint2006 MPKI by 25.6% from M1 to M6.

V. BRANCH PREDICTION SECURITY

During the evolution of the microarchitectures discussed here, several security vulnerabilities, including Spectre [21], became concerning. Several features were added to mitigate security holes. In this paper's discussion, the threat model is based on a fully-trustworthy operating system (and hypervisor if present), but untrusted userland programs, and that userland programs can create arbitrary code, whether from having full access, or from ROP/widget programming.

This paper only discusses features used to harden indirect and return stack predictions. Simple options such as erasing all branch prediction state on a context change may be necessary in some context transitions, but come at the cost of having to retrain when going back to the original context. Separating storage per context or tagging entries by context come at a significant area cost. The compromise solution discussed next provides improved security with minimal performance, timing, and area impact.

The new front-end mechanism hashes per-context state and scrambles the learned instruction address targets stored in a branch predictor's branch-target buffers (BTB) or return-address-stack (RAS). A mixture of software- and hardware-controlled entropy sources are used to generate the hash key (CONTEXT_HASH) for a process. The hashing of these stored instruction address targets will require the same exact numbers to perfectly un-hash and un-scramble the predicted taken target before directing the program address of the CPU. If a different context is used to read the structures, the learned target may be predicted taken, but will jump to an unknown/unpredictable address and a later mispredict recovery will be required. The computation of CONTEXT_HASH is shown in Figure 10.

The CONTEXT_HASH register is not software accessible, and contains the hash used for target encryption/decryption. Its value is calculated with several inputs, including:

- A software entropy source selected according to the user, kernel, hypervisor, or firmware level implemented as SCXTNUM_ELx as part of the security feature CSV2 (Cache Speculation Variant 2) described in ARM v8.5 [4].
- A hardware entropy source, again selected according to the user, kernel, hypervisor, or firmware level.
- Another hardware entropy source selected according to the security state.
- An entropy source from the combination of ASID (process ID), VMID (virtual machine ID), security state, and privilege level.

Note that the CONTEXT_HASH computation is performed completely in hardware, with no software visibility to intermediate values, even to the hypervisor.

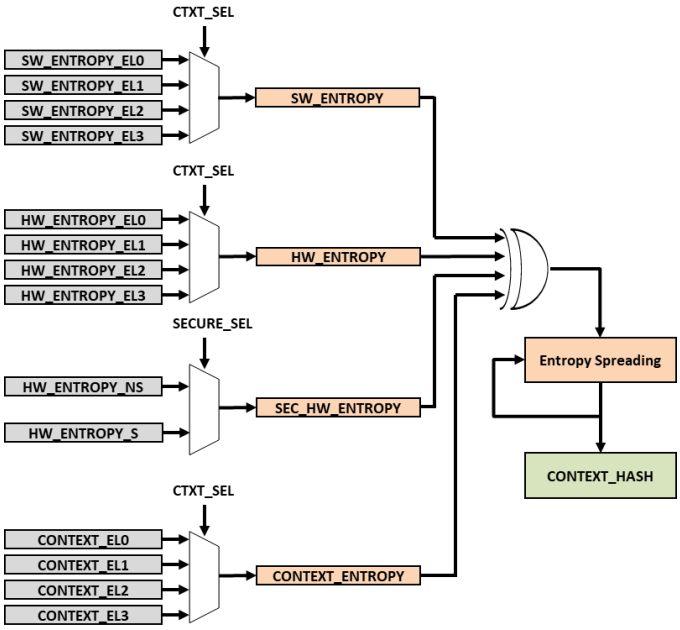

Fig. 10. Computation of CONTEXT_HASH encryption/decryption register [22]. This computation is only performed during a context switch, and takes only a few cycles.

The computation of the CONTEXT_HASH register also includes rounds of entropy diffusion [23] — specifically a deterministic, reversible non-linear transformation to average per-bit randomness. The small number of cycles required for the hardware to perform multiple levels of hashing and iterative entropy spreading have minimal performance impact, given the latency of a context switch.

Within a particular processor context, CONTEXT_HASH is used as a very fast stream cipher to XOR with the indirect branch or return targets being stored to the BTB or RAS, as in Figure 11. Such a simple cipher can be inserted into the RAS and BTB lookups without much impact to the timing paths, as compared to a more rigorous encryption/decryption scheme.

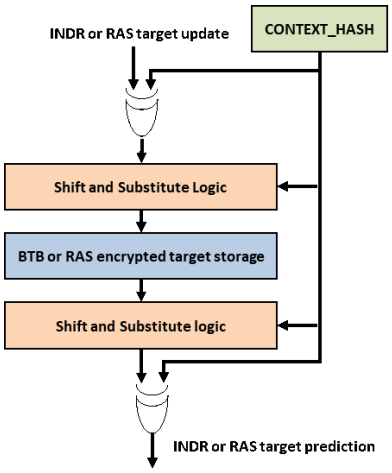

Fig. 11. Indirect/RAS Target Encryption [22]

To protect against a basic plaintext attack, a simple substitution cipher or bit reversal can further obfuscate the actual stored address. When the branch predictor is trained and ready to predict jump targets from these common structures, the hardware will use the program's CONTEXT_HASH to perfectly invert and translate out the correct prediction target.

This approach provides protection against both cross-training and replay attacks. If one attacker process tries to cross-train an indirect predictor with a target, it will be encrypted with $\text{CONTEXT\_HASH}_{\text{attack}}$ into the predictor structures, and then read out and decoded in the victim's execution with the differing $\text{CONTEXT\_HASH}_{\text{victim}}$, which will result in the victim beginning execution at a different address than the intended attack target. With regard to a replay attack, if one process is able to infer mappings from a plaintext target to a CONTEXT_HASH-encrypted target, this mapping will change on future executions, which will have a different process context (process ID, *etc*.).

In addition, the operating system can intentionally periodically alter the CONTEXT_HASH for a process or all processes (by changing one of the SW_ENTROPY_*_LVL inputs, for example), and, at the expense of indirect mispredicts and re-training, provide protection against undesired cross training during the lifetime of a process, similar to the concept in CEASER [24].

VI. Micro-Operation Cache (UOC)

M1 through M4 implementations fetched and decoded instructions as is common in a modern out-of-order pipeline: fetch instructions from the instruction cache, buffer them in the instruction queue, and generate micro-ops (µops) through the decoder before feeding them to later pipe stages. As the design moved from supplying 4 instructions/µops per cycle in M1, to 6 per cycle in M3 (with future ambitions to grow to 8 per cycle), fetch and decode power was a significant concern.

The M5 implementation added a micro-operation cache [25] [26] as an alternative µop supply path, primarily to save fetch and decode power on repeatable kernels. The UOC can hold up to 384 µops, and provides up to 6 µops per cycle to subsequent stages. The view of instructions by both the front-end and by the UOC is shown in Figure 12.

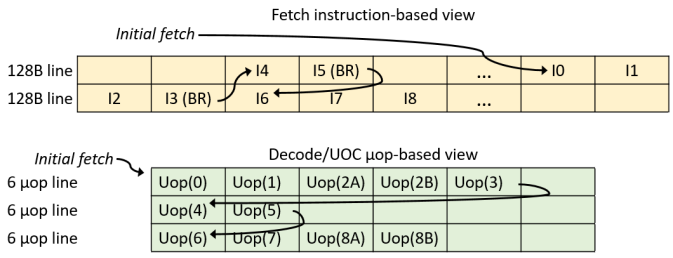

Fig. 12. Instruction-based and µop-based views

The front end operates in one of three different UOC modes: FilterMode, where the µBTB predictor determines predictability and size of a code segment; BuildMode, where the UOC is

allocating appropriate basic blocks; and FetchMode, where the instruction cache and decode logic are disabled, and instruction supply is solely handled by the UOC. Each mode is shown in the flowchart of Figure 13, and will be described in more detail in the next paragraphs.

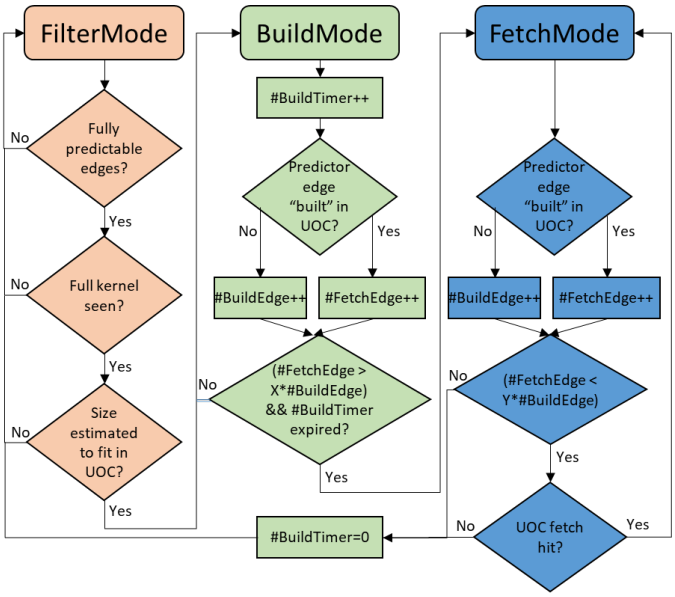

Fig. 13. Modes of UOC operation

The FilterMode is designed to avoid using the UOC BuildMode when it would not be profitable in terms of both power and performance. In FilterMode, the μBTB predictor checks several conditions to ensure the current code segment is highly predictable, and will also fit within both the μBTB and UOC finite resources. Once these conditions are met, the mechanism switches into BuildMode.

To support BuildMode, the μBTB predictor is augmented with "built" bits in each branch's prediction entry, tracking whether or not the target's basic block has already been seen in the UOC and back-propagated to the predictor. Initially, all of these "built" bits are clear. On a prediction lookup, the #BuildTimer is incremented, and the "built" bit is checked:

- If the "built" bit is clear, #BuildEdge is incremented, and the basic block is marked for allocation in the UOC. Next, the UOC tags are also checked ; if the basic block is present, information is propagated back to the μBTB predictor to update the "built" bit. This back-propagation avoids the latency of a tag check at prediction time, at the expense of an additional "build" request that will be squashed by the UOC.
- If the "built" bit is set, #FetchEdge is incremented.

When the ratio between #FetchEdge and #BuildEdge reaches a threshold, and the #BuildTimer has not timed out, it indicates that the code segment should now be mostly or completely UOC hits, and the front end shifts into FetchMode.

In FetchMode, the instruction cache and decode logic are disabled, and the μBTB predictions feed through the UAQ into the UOC to supply instructions to the rest of the machine. As long as the μBTB remains accurate, the mBTB is also disabled, saving additional power. While in FetchMode, the front end continues to examine the "built" bits, and updates #BuildEdge if the bit is clear, and #FetchEdge if the bit is set. If the ratio of #BuildEdge to #FetchEdge reaches a different threshold, it indicates that the current code is not hitting sufficiently in the UOC, and the front end shifts back into FilterMode.

VII. L1 Data Prefetching

Hardware prefetching into the L1 Cache allows data to be fetched from memory early enough to hide the memory latency from the program. Prefetching is performed based on information from demand loads, and can be limited by the size of the cache and maximum outstanding misses. Capacity of this cache grew from 32KB in M1, to 64KB in M3, to 128KB in M6. Outstanding misses grew from 8 in M1, to 12 in M3, to 32 in M4, and 40 in M6. The significant increase in misses in M4 was due to transitioning from a fill buffer approach to a data-less memory address buffer (MAB) approach that held fill data only in the data cache.

A. Algorithm

The L1 prefetcher detects strided patterns with multiple components (*e.g.* +1×3, +2×1, meaning a stride of 1 repeated 3 times, followed by a stride of two occuring only once). It operates on the virtual address space and prefetches are permitted to cross page boundaries. This approach allows large prefetch degrees (how many outstanding prefetches are allowed to be outstanding at a time) to help cover memory latency and solve prefetcher timeliness issues. This design also inherently acts as a simple TLB prefetcher to preload the translation of next pages prior to demand accesses to those pages.

The prefetcher trains on cache-misses to effectively use load pipe bandwidth. To avoid noisy behavior and improve pattern detection, out-of-order addresses generated from multiple load pipes are reordered back into program order using a ROB-like structure [27] [28]. To reduce the size of this re-order buffer, an address filter is used to deallocate duplicate entries to the same cache line. This further helps the training unit to see unique addresses.

The prefetcher training unit can train on multiple streams simultaneously and detect multi-strides, similar to [29]. If the same multi-stride pattern is repeated, it can detect a pattern. For example, consider this access pattern:

A; A+2; A+4; A+9; A+11; A+13; A+18 and so on

The above load stream has a stride of +2, +2, +5, +2, +2, +5. The prefetch engine locks onto a pattern of +2×2, +5×1 and generates prefetches A+20, A+22, A+27 and so on. The load stream from the re-order buffer is further used to confirm prefetch generation using a confirmation queue. Generated prefetch addresses get enqueued into the confirmation queue and subsequent demand memory accesses will match against the confirmation queue. A confidence scheme is used based on prefetch confirmations and memory sub-system latency to scale prefetcher degree (see next section for more details).

B. Dynamic Degree and One-pass/Two-pass

In order to cover the latency to main memory, the required degree can be very large (over 50). This creates two problems: lack of miss buffers, and excess prefetches for short-lived patterns. These excess prefetches waste power, bandwidth and cache capacity.

A new adaptive, dynamic degree [30] mechanism avoids excessive prefetches. Prefetches are grouped into windows, with the window size equal to the current degree. A newly created stream starts with a low degree. After some number of confirmations within the window, the degree will be increased. If there are too few confirmations in the window, the degree is decreased. In addition, if the demand stream overtakes the prefetch stream, the prefetch issue logic will skip ahead of the demand stream, avoiding redundant late prefetches.

To reduce pressure on L1 miss buffers, a "two pass mode" scheme [31] is used, starting with M1, as shown in Figure 14. Note that M3 and beyond added an L3 between the L2 and interconnect. When a prefetch is issued the first time, it will not allocate an L1 miss buffer, but instead be sent as a fill request into the L2 (1). The prefetch address will also be inserted (2) into a queue, which will be held until sufficient L1 miss buffers are available. The L2 cache will issue the request to the interconnect (3) and eventually receive a response (4). When a free L1 miss buffer is available, an L1 miss buffer will allocate (5) and an L1 fill request will occur (6 and 7).

This scheme is suboptimal for cases where the working set fits in the L2 (since every first pass prefetch to the L2 would hit). To counteract this, the mechanism also tracks the number of first pass prefetch hits in the L2, and if they reach a certain watermark, it will switch into "one pass mode". In this mode, shown on the right in Figure 14, only step 2 is initially performed. When there are available miss buffers (which may be immediately), steps 5, 6, and 7 are performed, saving both power and L2 bandwidth.

C. Spatial Memory Streaming Prefetcher

The multi-strided prefetcher is excellent at capturing regular accesses to memory. However, programs which traverse a linked-list or other certain types of data structures are not covered at all. To attack these cases, in M3 an additional L1 prefetch engine is added — a spatial memory stream (SMS) prefetcher [32] [33]. This engine tracks a primary load (the first miss to a region), and attaches associated accesses to it (any misses with a different PC). When the primary load PC appears again, prefetches for the associated loads will be generated based off the remembered offsets.

Also, this SMS algorithm tracks confidence for each associated demand load. Only associated loads with high confidence are prefetched, to filter out the ones that will appear transiently along with the primary load. In addition, when confidence drops to a lower level, the mechanism will only issue the first pass (L2) prefetch.

With two different prefetch engines, a scheme is required to avoid duplicate prefetches. Given that a trained multi-stride engine can prefetch further ahead of the demand stream than the SMS engine, confirmations from the multi-stride engine suppress training in the SMS engine.

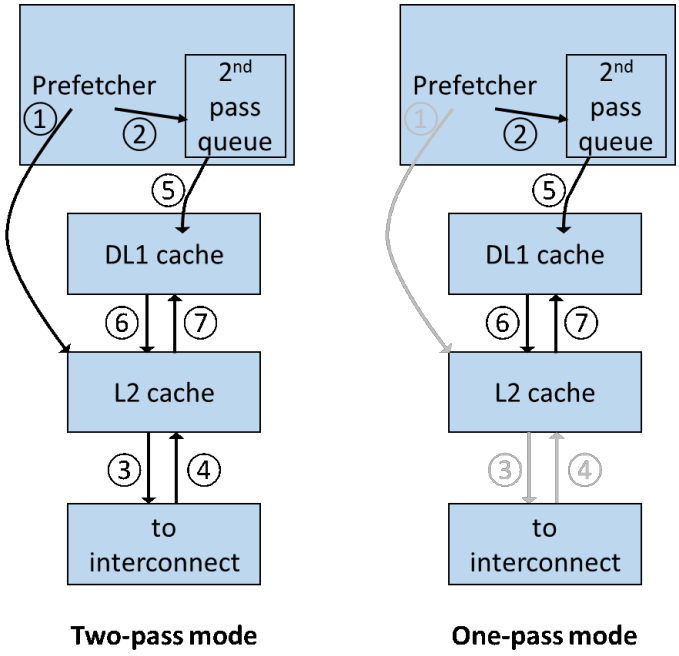

Fig. 14. One-pass/two-pass prefetching scheme for M1/M2.

D. Integrated Confirmation Queues

To cover memory latency and allow multiple load streams to operate simultaneously, the size of the confirmation queue needs to be considerably large. Also, at the start of pattern detection, the prefetch generation can lag behind the demand stream. Since confirmations are based on issued prefetches, this can result in few confirmations. As prefetch degree is a function of confirmations, this mechanism can keep the degree in a low state and prefetches may never get ahead of the demand stream.

To combat the above issues, the M3 core introduced an integrated confirmation scheme [34] to replace the confirmation queue. The new mechanism keeps the last confirmed address, and uses the locked prefetch pattern to generate the next N confirmation addresses into a queue, where N is much less than the stream's degree. This is the same logic as the prefetch generation logic, but acts independently. This confirmation scheme reduces the confirmation structure size considerably and also allows confirmations even when the prefetch engine has not yet generated prefetches.

VIII. Large Cache Management and Prefetchers

Over six generations, the large cache (L2/L3) hierarchy evolved as shown in Table III below. M3 added a larger but more complex three-level hierarchy, which helps balance latency versus capacity objectives with the increasing working set needs of modern workloads.

TABLE III
EVOLUTION OF CACHE HIERARCHY SIZES

	L2 Cache	L3 Cache
M1/M2	2MB	–
M3	512KB	4MB
M4	1MB	3MB
M5	2MB	3MB
M6	2MB	4MB

A. Co-ordinated Cache Hierarchy Management

To avoid data duplication, the outer cache levels (L3) are made exclusive to the inner caches (L1 and L2). Conventional exclusive caches do not keep track of data reuse since cache lines are swapped back to inner-level caches. The L2 cache tracks both frequency of hits within the L2, as well as subsequent re-allocation from the L3 cache. Upon L2 castout, these details are used to intelligently choose to either allocate the castouts into the L3 in an elevated replacement state, or an ordinary replacement state, or avoid allocation altogether.

This bidirectional coordination allows the caches to preserve useful data in the wake of transient streams, or when the application working set exceeds the total cache capacity. Amongst the hardware overhead, each cache tag stores some meta-data to indicate the reuse or dead behavior of the associated lines, and for different transaction types (prefetch vs. demand). This meta-data is passed through request or response channels between the cache levels. There are some cases that needed to be filtered out from being marked as reuse, such as the second pass prefetch of two-pass prefetching. More information can be found in the related patent [35].

B. Buddy Cache Prefetcher

The L2 cache tags are sectored at a 128B granule for a default data line size of 64B. This sectoring reduces the tag area and allows a lower latency for tag lookups. Starting in M4, a simple "Buddy" prefetcher is added that, for every demand miss, generates a prefetch for its 64B neighbor (buddy) sector. Due to the tag sectoring, this prefetching does not cause any cache pollution, since the buddy sector will stay invalid in absence of buddy prefetching. There can be an impact on DRAM bandwidth though, if the buddy prefetches are never accessed. To alleviate that issue, a filter is added to the Buddy prefetcher to track the patterns of demand accesses. In the case where access patterns are observed to almost always skip the neighboring sector, the buddy prefetching is disabled.

C. Standalone Cache Prefetcher

Starting in M5, a standalone prefetcher is added to prefetch into the lower level caches beyond the L1s. This prefetcher observes a global view of both the instruction and data accesses at the lower cache level, to detect stream patterns [36]. Both demand accesses and core-initiated prefetches are used for its training. Including the core prefetches improves their timeliness when they are eventually filled into the L1. Among the challenges of the standalone prefetcher is the out-of-orderness of program accesses observed at the lower-level cache that can pollute the patterns being trained. Another challenge is the fact that it operates on physical addresses, which limits its span to a single page. Similarly, hits in the L1 can filter the access stream seen by the lower levels of cache, making it difficult to isolate the real program access stream. For those reasons, the standalone prefetcher uses an algorithm to handle long, complex streams with larger training structures, and techniques to reuse learnings across 4KB physical page crossings. The standalone prefetcher also employs a two-level adaptive scheme to maintain high accuracy of issued prefetches as explained below.

D. Adaptive Prefetching

The standalone prefetcher is built with an adaptive scheme shown in Figure 15 with two modes of prefetching: low confidence mode and high confidence mode. In low confidence mode, "phantom" prefetches are generated for confidence tracking purposes into a prefetch filter, but not issued to the memory system or issued very conservatively. The confidence is tracked through demand accesses matching entries in the prefetch filter. When confidence increases beyond a threshold, high confidence mode is entered.

In high confidence mode, prefetches are issued aggressively out to the memory system. The high-confidence mode relies on cache meta-data to track the prefetcher's confidence. This meta-data includes bits associated with the tags to mark whether a line was prefetched, and if it was accessed by a demand request. If the confidence reduces below a threshold, the prefetcher transitions back to the low confidence mode. Hence, the prefetcher accuracy is continuously monitored to detect transitions between application phases that are prefetcher friendly and phases that are difficult to prefetch or when the out-of-orderness makes it hard to detect the correct patterns.

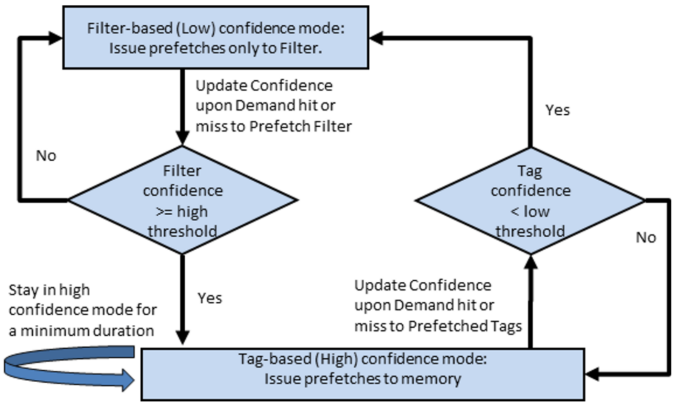

Fig. 15. Adaptive Prefetcher State Transitions

IX. MEMORY ACCESS LATENCY OPTIMIZATION

The Exynos mobile processor designs contain three different voltage/frequency domains along the core's path to main memory: the core domain, an interconnect domain, and a memory controller domain. This provides flexibility with

regards to power usage in different mobile scenarios: GPU-heavy, core-heavy, etc. However, this requires four on-die asynchronous crossings (two outbound, two inbound), as well as several blocks' worth of buffering, along that path. Over several generations, the Exynos designs add several features to reduce the DRAM latency through multiple levels of the cache hierarchy and interconnect. These features include data fast path bypass, read speculation, and early page activate.

The CPU memory system on M4 supports an additional dedicated data fast path bypass directly from DRAM to the CPU cluster. The data bypasses multiple levels of cache return path and interconnect queuing stages. Also a single direct asynchronous crossing from the memory controller to the CPU bypasses the interconnect domain's two asynchronous crossings.

To further optimize latency, the CPU memory system on M5 supports speculative cache lookup bypass for latency critical reads. The read requests are classified as "latency critical" based on various heuristics from the CPU (*e.g.* demand load miss, instruction cache miss, table walk requests *etc.*) as well as a history-based cache miss predictor. Such reads speculatively issue to the coherent interconnect in parallel to checking the tags of the levels of cache. The coherent interconnect contains a snoop filter directory that is normally looked up in the path to memory access for coherency management. The speculative read feature utilizes the directory lookup [37] to further predict with high probability whether the requested cache line may be present in the bypassed lower levels of cache. If yes, then it cancels the speculative request by informing the requester. This cancel mechanism avoids penalizing memory bandwidth and power on unnecessary accesses, acting as a second-chance "corrector predictor" in case the cache miss prediction from the first predictor is wrong.

The M5 CPU memory system contains another feature to reduce memory latency. For latency critical reads, a dedicated sideband interface sends an early page activate command to the memory controller to speculatively open a new DRAM page. This interface, like the fast data path mechanism described above, also bypasses two asynchronous crossings with one. The page activation command is a hint the memory controller may ignore under heavy load.

X. Overall Impact on Average Load Latency

The net impact of all of the generational changes on average load latency is shown in Figure 16. These include the cache size, prefetching, replacement algorithms, and DRAM latency changes discussed in the above sections. Note that the 3-cycle cascading load latency feature is clearly visible on the left of the graph for workloads that hit in the DL1 cache. It also partially contributes to lower DL1 cache hit latencies for many other workloads that occasionally miss. There are many other changes not discussed in this paper that also contributed to lower average load latency.

Overall, these changes reduce average load latency from 14.9 cycles in M1 to 8.3 cycles in M6, as shown in Table IV.

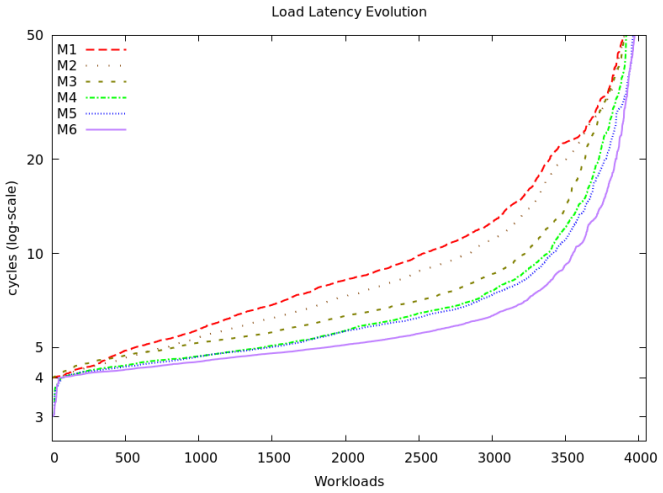

Fig. 16. Evolution of Average Load Latency

TABLE IV
GENERATIONAL AVERAGE LOAD LATENCIES

	M1	M2	M3	M4	M5	M6
Avg. load lat.	14.9	13.8	12.8	11.1	9.5	8.3

XI. Conclusion

This paper discusses the evolution of a branch predictor implementation, security protections, prefetching improvements, and main memory latency reductions. These changes, as well as dozens of other features throughout the microarchitecture, resulted in substantial year-on-year frequency-neutral performance improvements.

Throughout the generations of the Exynos cores, the focus was to improve performance across all types of workloads, as shown in Figure 17:

- Low-IPC workloads were greatly improved by more sophisticated, coordinated prefetching, as well as cache replacement/victimization optimizations.
- Medium-IPC workloads benefited from MPKI reduction, cache improvements, additional resources, and other performance tweaks.
- High-IPC workloads were capped by M1's 4-wide design. M3 and beyond were augmented in terms of width and associated resources needed to achieve 6 IPC for workloads capable of that level of parallelism.

The average IPC across these workloads for M1 is 1.06, while the average IPC for M6 is 2.71, which showcases a compounded growth rate of 20.6% frequency-neutral IPC improvement every year.

Acknowledgments

The work described here represents the combined effort of hundreds of engineers over eight years of innovation and hard work in order to meet all of the schedule, performance, timing, and power goals, for five annual generations of productized silicon and a sixth completed design. The authors are deeply

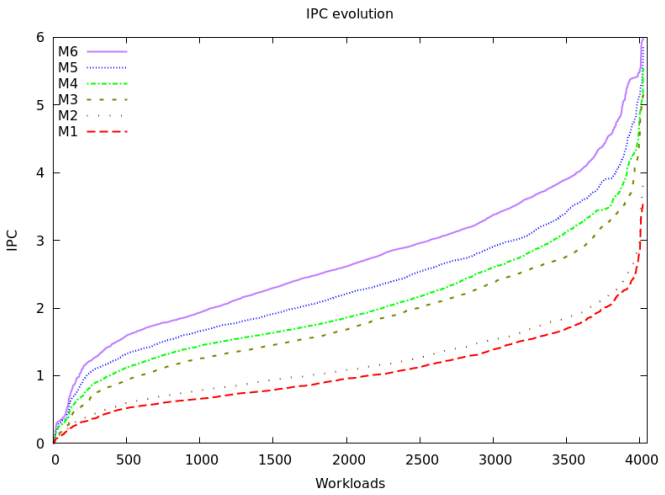

Fig. 17. IPCs across 4026 workload slices, by generation

indebted to Samsung for supporting the publication of this paper, and to Keith Hawkins, Mike Goddard, and Brad Burgess for their leadership of SARC during these designs. The authors also thank the anonymous reviewers for their comments and suggestions on improvements.

REFERENCES

[1] B. Burgess, "Samsung Exynos M1 Processor," in IEEE Hot Chips 28, 2016.
[2] J. Rupley, "Samsung Exynos M3 Processor," in IEEE Hot Chips 30, 2018.
[3] J. Rupley, B. Burgess, B. Grayson and G. Zuraski, "Samsung M3 Processor," IEEE Micro, Vol 39, March-April 2019.
[4] ARM Incorporated, "ARM Architectural Reference Manual ARMv8, for Armv8-A architectural profile," Revision E.a, 2019.
[5] E. Perelman, G. Hamerly, M. V. Biesbrouck, T. Sherwood and B. Calder, "Using SimPoint for Accurate and Efficient Simulation," in ACM SIGMETRICS Performance Evaluation Review, 2003.
[6] D. A. Jiménez and C. Lin, "Dynamic Branch Prediction with Perceptrons," in High-Performance Computer Architecture (HPCA), 2001.
[7] D. A. Jiménez, "Fast Path-Based Neural Branch Prediction," in Proceedings of the 36th Annual International Symposium on Microarchitecture (MICRO-36), 2003.
[8] D. A. Jiménez, "An Optimized Scaled Neural Branch Predictor," in IEEE 29th International Conference on Computer Design (ICCD), 2011.
[9] D. A. Jiménez, "Strided Sampling Hashed Perceptron Predictor," in Championship Branch Prediction (CBP-4), 2014.
[10] D. Tarjan and K. Skadron, "Merging Path and Gshare Indexing in Perceptron Branch Prediction," Transactions on Architecture and Code Optimization, 2005.
[11] S. McFarling, "Combining Branch Predictors. TN-36," Digital Equipment Corporation, Western Research Laboratory, 1993.
[12] T.-Y. Yeh and Y. N. Patt, "Two-level Adaptive Branch Prediction," in 24th ACM/IEEE International Symposium on Microarchitecture (MICRO-24), 1991.
[13] R. Nair, "Dynamic Path-based Branch Correlation," in 28th Annual International Symposium on Microarchitecture (MICRO-28), 1995.
[14] "CBP-5 Kit," in 5th Championship Branch Prediction Workshop (CBP-5), 2016.
[15] A. Seznec, "Analysis of the O-GEometric History Length Branch Predictor," in 32nd International Symposium on Computer Architecture (ISCA-32), 2005.
[16] P.-Y. Chang, M. Evers and Y. N. Patt, "Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference," in Conference on Parallel Architectures and Compilation Technique (PACT), 1996.
[17] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt and R. Cohn, "Virtual Program Counter (VPC) Prediction: Very Low Cost Indirect Branch Prediction Using Conditional Branch Prediction Hardware," IEEE Transactions on Computers, vol. 58, no. 9, 2009.
[18] J. Dundas and G. Zuraski, "High Performance Zero Bubble Conditional Branch Prediction using Micro Branch Target Buffer," United States Patent 10,402,200, 2019.
[19] E. Jacobson, E. Rotenberg and J. E. Smith, "Assigning Confidence to Conditional Branch Predictions," in 29th International Symposium on Microarchitecture (MICRO-29), 1996.
[20] R. Jumani, F. Zou, M. Tkaczyk and E. Quinnell, "Mispredict Recovery Apparatus and Method for Branch and Fetch Pipelines," Patent Pending.
[21] P. Kocher, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in IEEE Symposium on Security and Privacy, 2018.
[22] M. Tkaczyk, E. Quinnell, B. Grayson, B. Burgess and M. B. Barakat, "Secure Branch Predictor with Context-Specific Learned Instruction Target Address Encryption," Patent Pending.
[23] C. E. Shannon, "A Mathematical Theory of Cryptography," Bell System Technical Memo MM 45-110-02, September 1, 1945.
[24] M. K. Qureshi, "CEASER: mitigating conflict-based cache attacks via encrypted-address and remapping," in Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-51), 2018.
[25] B. Solomon, A. Mendelson, D. Orenstien, Y. Almog and R. Ronen, "Micro-Operation Cache: A Power Aware Frontend for Variable Instruction Length ISA," in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, 2003.
[26] E. Quinnell, R. Hensley, M. Tkaczyk, J. Dundas, M. S. S. Govindan and F. Zou, "Micro-operation Cache Using Predictive Allocation," Patent Pending.
[27] J. Smith and A. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors," in 12th Annual International Symposium on Computer Architecture (ISCA-12), 1985.
[28] A. Radhakrishnan and K. Sundaram, "Address Re-ordering Mechanism for Efficient Pre-fetch Training in an Out-of order Processor," United States Patent 10,031,851, 2018.
[29] S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou and S. G. Abraham, "Effective stream-based and execution-based data prefetching," in Proceedings of the 18th Annual International Conference on Supercomputing (ICS-18), 2004.
[30] A. Radhakrishnan and K. Sundaram, "Adaptive Mechanism to tune the Degree of Pre-fetches Streams," United States Patent 9,665,491, 2017.
[31] A. Radhakrishnan, K. Lepak, R. Gopal, M. Chinnakonda, K. Sundaram and B. Grayson, "Pre-fetch Chaining," United States Patent 9,569,361, 2017.
[32] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi and A. Moshovos, "Spatial Memory Streaming," in 33rd International Symposium on Computer Architecture (ISCA-33), 2006.
[33] E. A. Brekelbaum and A. Radhakrishnan, "System and Method for Spatial Memory Streaming Training," United States Patent 10,417,130, 2019.
[34] E. Brekelbaum and A. Ghiya, "Integrated Confirmation Queues," United States Patent 10,387,320, 2019.
[35] Y. Tian, T. Nakra, K. Nguyen, R. Reddy and E. Silvera, "Coordinated Cache Management Policy for an Exclusive Cache Hierarchy." Patent patent pending, 2017.
[36] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson and Z. Chishti, "Path confidence based lookahead prefetching," in Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49), 2016.
[37] V. Sinha, H. Le, T. Nakra, Y. Tian, A. Patel and O. Torres, "Speculative DRAM Read, in Parallel with Cache Level Search, Leveraging Interconnect Directory," Patent Pending.