2018 51st Annual IEEE/ACM International Symposium on Microarchitecture

CritICs Critiquing Criticality in Mobile Apps

Prasanna Venkatesh Rengasamy*, Haibo Zhang*, Shulin Zhao*, Nachiappan Chidambaram Nachiappan‡, Anand Sivasubramaniam*, Mahmut T Kandemir*, Chita R Das*
*The Pennsylvania State University,‡IEEE Member
Email:{pur128,haibo,suz53,anand,kandemir,das}@psu.edu, nachi@alumni.psu.edu

***Abstract*—In this paper, we conduct a systematic analysis to show that existing CPU optimizations targeting scientific/server workloads are not always well suited for mobile apps. In particular, we observe that the well-known and very important concept of identifying and accelerating individual critical instructions in workloads such as SPEC, are not as effective for mobile apps. Several differences in mobile app characteristics including (i) dependencies between critical instructions interspersed with non-critical instructions in the dependence chain, (ii) temporal proximity of the critical instructions in the dynamic stream, and (iii) the bottleneck shifting to the front from the rear of the datapath pipeline, are key contributors to the ineffectiveness of traditional criticality based optimizations. Instead, we propose the concept of *Critical Instruction Chains* (CritICs) – which are short, critical and self contained sequences of instructions, for aggregate level optimization. With motivating results, we show that an offline profiler/analysis framework can easily identify these CritICs, and we propose a very simple software mechanism in the compiler that exploits ARM's 16-bit ISA format to nearly double the fetch bandwidth of these instructions. We have implemented this entire framework - both profiler and compiler passes, and evaluated its effectiveness for 10 popular apps from the Play Store. Experimental evaluations show that our approach is much more effective than two previously studied criticality optimizations, yielding a speedup of 12.65%, and energy savings of 15% in the CPU (translating to a system wide energy savings of 4.6%), requiring very little additional hardware support.**

***Index Terms*—Criticality, CPU, Mobile, Energy**

I. Introduction

The proliferation of mobile devices over the past decade has been fueled by not just hardware advancements, but also by the numerous and diverse applications (apps) that these devices can support. The number of such devices far exceeds the desktop and server markets, with nearly 2.6 billion mobile devices serving more than 35% of the world population today [1]–[3]. To a large extent, the hardware and software evolution of these devices has drawn from lessons learned over the years from their desktop/server counterparts and adapted them for different resource constraints – energy/power, form-factor, etc. On the other hand, many of the mobile apps have very different characteristics, and are used in very different ways compared to desktop/server workloads (e.g., high amount of user-interaction, handling sensors, etc.). And so, it is not clear whether the same high-end device optimizations are effective for the mobile platforms.

Picking two well-studied optimizations (instruction prioritization and memory prefetching) that try to exploit a very important property, namely "criticality", of the instructions in server/desktop domains, this paper points out that these mechanisms are not well suited to mobile apps. Instead, this work proposes to track criticality at the granularity of self-contained instruction chains, and assigns a criticality metric to each chain. With the bottleneck shifting from the rear to the front of the CPU datapath pipeline in these mobile apps, this paper introduces a novel way of prioritizing and aggregating these Critical Instruction Chains (`CritIC`) in software to solve this problem, requiring little to no additional hardware support.

While one could throw extra resources and hardware mechanisms into a superscalar processor's datapath to boost performance in embedded domain, it is even more important to better utilize the existing resources amongst the competing instructions, especially in resource-constrained environments. One such well studied mechanism for prioritization amongst competing instructions is based on "criticality". When a critical instruction is brought into the processor datapath, different prioritizations/optimizations can be employed. Over the years, numerous criticality based optimizations have been proposed and studied for high-end workloads - prioritizing CPU resources [4]–[7], caches [8]–[10], memory request queues [11]–[13], predicting the result of the instruction [14]–[17], issuing prefetch requests [18], etc. However, the impact of these optimizations has not been studied to date for mobile workloads, and that is one important void this paper intends to fill.

Unlike many desktop/server workloads which are very throughput demanding, mobile apps are highly user-interactive. User actions like screen swipes, and sensor inputs like positional (GPS) and/or movement (accelerometer) frequently control the execution, with the app reacting to such actions and coming back for additional inputs. The consequent code, though rich in the conventionally deemed "critical instructions", are not conducive enough for independent instruction-level optimizations, i.e., they are often dependent on each other with possibly one or more non-critical instructions coming into the dependence chain between them. For instance, we show that one such recent optimization [18], which prioritizes critical loads, does very well for SPEC workloads (as in prior works), but provides a measly 0.7% speedup for a wide spectrum of mobile apps. Any criticality optimization targeting mobile apps should, thus, consider these sequences/groups of critical instructions at a time, rather than optimize for each individually. We identify two additional differences: (i) the bottleneck for these critical instructions shifts from the back-end (Execute/Commit stages of the superscalar pipeline) to the front-end (Fetch stage) of the pipeline when we move from SPEC to mobile apps; and (ii) sequences/groups of critical instructions are much smaller, and occur close together in the dynamic execution stream, for mobile apps compared to SPEC workloads, making them more amenable for aggregate-level software based optimizations. We are not aware of any prior work which has pointed out the insufficiencies of criticality-based individual instruction optimizations for mobile apps, and their workload differences

978-1-5386-6240-3/18/$31.00 ©2018 IEEE
DOI 10.1109/MICRO.2018.00075

requiring a revisit of this topic in the mobile context.

Motivated by this insight for several off-the-shelf Android apps, we make the following contributions in this work:

- We use the concept of a self-contained Instruction Chain (IC) within a Data Flow Graph (DFG), which can be executed independent of other instructions, when encountered. We introduce a criticality metric for an IC, that is calculated as the average fan-out per instruction in that chain. ICs with a criticality exceeding a threshold are marked as CritICs, and the entire CritIC sequence of instructions is given priority. Unlike SPEC, CritICs in mobile apps are relatively short (order of 5 instructions) and are not that widely spaced out in the dynamic instruction stream either, making them suitable for software (e.g., compiler, profiler) identification.
- We note that CritIC instructions in mobile apps are bottlenecked in the Fetch stage of the pipeline, as opposed to SPEC which are back-ended (execute/commit stages), since the number of high latency instruction is a much smaller fraction in the former. Both the producer side which feeds instructions into the pipeline, and the consumer side which drains the instructions from the fetched queue are equally important contributors to this bottlenecked stage.
- Our identification of Critical and Self-contained ICs, provides a convenient abstraction for tackling the fetch bottleneck. We could theoretically hoist and aggregate all these instructions together as a macro instruction. But the number of possible CritIC sequences makes this option expensive. Adhering to the philosophy of imposing minimal hardware enhancements, we instead propose a novel approach to doubling the fetch bandwidth for these CritIC instructions by leveraging ARM's 16-bit instruction format. We convert all these instructions into the 16 bit representation in a compiler pass, as long as there is no loss in their functionality, together with a preceding command to instruct the ARM hardware to switch to 16-bit format for these instructions. This proposed hoisting and 16-bit conversion from the software side can be employed in all current ARM CPU based devices [19]–[21].
- We have implemented a profiler on top of the AOSP emulator [22] and GEM5 simulator [23] for identifying CritIC sequences. We have also added a compiler pass in the Android Runtime Compiler (ART) to hoist, aggregate and emit the 16-bit representations for the CritIC instructions. We have evaluated our proposal in GEM5 for a Google Tablet configuration using a diverse and popular suite of 10 stock Android apps from the Play Store.
- Results show that our approach provides as much as 15% speedup (12.65% speedup on the average) for these apps, while two previously well regarded single instruction criticality based load prefetching and single instruction prioritization provide only 0.7% and 5% speedup on the average. The 16-bit representations nearly doubles the fetch bandwidth for all CritIC sequences, buying back 3.6% of the time on the producer-side of the fetch. Packing them back-to-back reduces the data flow time across these instructions, buying back 2.5% of the time on the consumer-side of the fetch. All this is achieved with little to no extra hardware requirements in the existing processor datapath.
- While one could increase i-cache capacity, improve branch prediction accuracies, and/or employ instruction prefetchers to address the fetch bottleneck, all these may require extra hardware than what current mobile platforms support because of resource-constraints. Even if future platforms incorporate such additional hardware to address the fetch problem, we show that our simple software technique can do as well as any of these hardware approaches (even a platform that has 4× the i-cache capacity and a perfect branch predictor). Further, it can synergistically provide an additional 11% speedup over what these hardware techniques can provide.
- It may appear that one could opportunistically use the 16-bit ARM format for all instructions to optimize the fetch stage for all instructions. We show that while this can provide 6% improvement, our approach of identifying, hoisting and grouping CritIC sequences, and selectively using the format for such instruction sequences, does much better (12.65%). In fact, using our solution, and then opportunistically using the 16-bit format for other instructions, complement each other to provide 16% improvement.

II. Critiquing Criticality

Within the confines of the given resources of a superscalar processor, one of the most important issues is deploying and assigning these resources to the incoming stream of instructions. This is essentially determined by the priority order (scheduling) for fetching and executing these instructions. When there are adequate resources, we would give all instructions the resources that they need. However, when resources are constrained, priority has to be given to "critical instructions" [4], [9], [12], [24]–[26]. In this work, we use a simple definition of criticality, similar to those in some prior works [4], [26] - an instruction is critical if its execution time becomes visible (i.e., does not get hidden) in the overall app execution.

A. Conventional criticality identification

As per the above definition, an instruction can be marked critical, only after its execution - by which time it is too late to assign resources for it. Hence, prior works propose different ways of estimating criticality of an instruction before it is even fetched. Two common heuristics for marking an instruction as critical are by using thresholds for (i) execution latency of an instruction (a long latency instruction implies

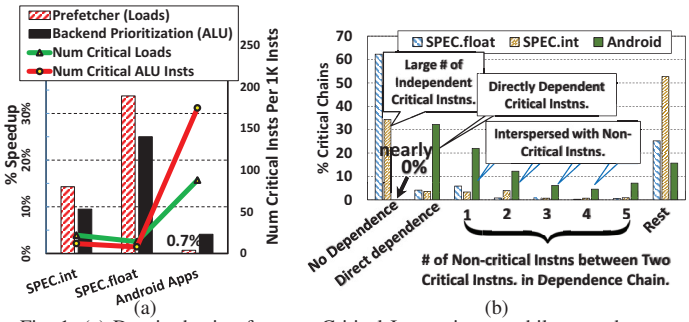
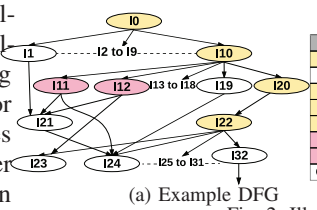


Fig. 1: (a) Despite having frequent Critical Instructions, mobile apps do not benefit as much. (b) Reason: Critical instructions in SPEC do not depend much on other critical instructions. But, Android apps have two successive high-fanout instructions in a dependence chain, with 0(direct-dependence) to 5 low fanout instructions between them.

instructions depending on it have to be delayed, thus making it more critical) [8], [9], [26] and (ii) number of dependent instructions (referred to as *fanout* in this paper), particularly in the ROB at the time the instruction is being executed (as many instructions require its output before they can begin). A table is maintained for those instructions exceeding the threshold based on prior execution (similar to branch predictors), and upon an instruction fetch, this table is looked up with the PC to find whether that instruction is critical or not.

B. Do these criticality schemes work for mobile apps?

Different optimizations can be employed upon fetching a critical instruction - prioritizing CPU resources [26]–[28], caches [8], [9], [18], memory requests [11], predicting instruction results [14], [29]–[31], issuing prefetches [18], etc. Until now, these optimizations have been primarily proposed and evaluated for server/desktop workloads and not for mobile apps/platforms. Without loss in generality, we have taken two representative, well-studied and well-proven criticality optimizations in prioritizing two important resources - one for memory which issues prefetches for critical loads [18] and another for ALU resources in instruction scheduling [4], [32], [33]. These proposals identify high-fanout loads to mark them as critical to issue prefetch [18] and prioritize the critical instructions for ALU resource allocations [32], [33]. These techniques have shown significant benefits for server workloads. The high-fanout based optimization has also been shown to outperform the latency based ways of identifying and exploiting criticality [18], [34]. We next evaluate the usefulness of both these criticality optimizations (depicted as bars) in mobile apps and compare the mean speedup obtained from employing both these techniques for SPEC.int, SPEC.float and Android apps in Fig. 1a (experimental details are in Sec. IV-B).



(a) Example DFG

Fanout Data

Ins	Fanout
I0	10
I1	1
I10	10
I20	1
I22	10
I11	2
I12	2
Others	1

High Fanout Instruction Optimization: (Conventional)

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Num Issues	1	2	2	2	2	2	2	2	2	2	2	1	1	2	2	2	2	2
Inst. [1]	I0	I10	I11	I3	I5	I7	I9	I14	I16	I1	I19	I20	I22	I23	I25	I27	I29	I31
Inst. [2]		I2	I12	I4	I6	I8	I13	I15	I17	I18	I21			I24	I26	I28	I30	I32

IC-Based Optimization: (Our Scheme)

Cycles	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Num Issues	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
Inst. [1]	I0	I10	I20	I22	I4	I6	I8	I11	I13	I15	I17	I19	I23	I25	I27	I29	I31
Inst. [2]		I1	I2	I3	I5	I7	I9	I12	I14	I16	I18	I21	I24	I26	I28	I30	I32

(b) Timing in superscalar processor with issue width 2

Fig. 2: Illustrating why high-fanout prioritization may not help.

As can be seen, the performance gains from prefetching high-fanout loads and prioritizing them at ALU resource scheduling are both quite significant for SPEC.int (15% from prefetching, 9% from prioritizing) and SPEC.float (34% from prefetching, 25% from prioritizing), re-affirming prior results [18], [33]. Interestingly, the gains from these two optimizations are a relatively measly 0.7% from prefetching and 5% from prioritizing in the mobile apps. Based on this, one may think that perhaps mobile apps do not have a significant number of high fanout loads/ALU instructions to benefit from these optimizations. On the contrary, we observe that (in right y-axis of Fig. 1a) the mobile apps have a much higher percentage of critical instructions than their SPEC counterparts. This should have, in turn, resulted in more opportunities for optimizing the execution. To understand why this is not the case, we next identify scenarios where these optimizations may not work and point out that such scenarios are common in mobile apps.

C. Why do they not work?

Fig. 2a shows an example DFG (Directed Flow Graph) where, executing the first instruction I0, triggers ten following instructions (I1 to I10) to become ready for execution. Any high fanout optimization will obviously execute I0 first. After this step, let us say I10 again has a fanout of 10 (i.e., instructions I11 to I20 become ready), which would cause I10 to be prioritized in the execution over say I1. If, subsequently, I11 and I12 each have 2 fanouts and each of I13 to I20 has a fanout of just 1, I11 and I12 will get scheduled before I13 to I20. But since each of I13 to I20 instructions has a fanout of 1, a high-fanout instruction prioritization will not differentiate between them. Note that, I20 in turn has a dependent high fanout instruction, I22, that cannot be scheduled till I20 is completed. So, as seen in Fig. 2b, by not doing this optimization of prioritizing I20 over its siblings, single a instruction criticality optimization scheme as described previously, stalls 2 cycles (12, 13) in the execution. This scenario occurs commonly in mobile executions (explained below), where an instruction despite having a low fanout, requires high-priority since there is a subsequently dependent high-fanout instruction. Consequently, it is insufficient to optimize individual high-fanout instructions independently. Instead, the whole sequence of dependent instructions from ⟨I0, I10, I20 to I22⟩ should be scheduled as early as possible, even though I20 is a low-fanout instruction. We find evidence of this scenario occurring much more in mobile apps compared to their SPEC counterparts as shown in Fig. 1b, which breaks down the dependence chains containing high-fanout instructions in terms of the number of low-fanout instructions between two successive high fanout instructions in a dependence chain. We find that the dependence chains can have between 1(22%) to 5(7%) low-fanout instructions in the dependence chain between two high fanout critical instructions, for cumulatively 52% of the time in Android apps. On the other hand, the SPEC.float and SPEC.int apps have no dependent high-fanout instructions for around 60% and 35% of the time. Compare that to Android apps, where this hardly ever happens, i.e., there is at least 1 low fanout instruction between 2 successive high fanout, and thereby critical ones. It is no surprise that SPEC apps benefited from optimizing each critical instruction individually as opposed to Android ones, where such dependent chains reduce the effectiveness of individual optimizations. These mobile app results also suggest that: (a)prioritizing/optimizing each critical instruction individually as it comes (i.e., for the "present") would not be as effective in rightfully apportioning the given resources; and (b)we need to consider these temporally proximate and dependent critical instructions (chains/sequences) together for possible optimizations, i.e., look into the future as well. Traditional criticality based optimizations [9], [11], [12], [18] have targeted one critical instruction at a time, rather than groups or chains.

D. What do these instructions need?

Before optimizing for these closely occurring and dependent critical instructions in Android apps, it is important to understand where they spend their time amongst the different superscalar pipeline stages. Towards this, we present a breakdown of their execution profiles amongst these stages in Fig. 3(a). In the same graph, we provide a similar profile for critical instructions identified by the same "high fanout" metric in the SPEC.float and SPEC.int apps. From these results, we observe the following: (i) unlike SPEC apps, where the Execute stage, and consequently the back-pressure in ROB queue residencies, are quite dominant, the Android apps have a much lower Execution stage latency (and consequently the ROB residency). The mix of critical instructions in Android apps do not take as much execution time (fewer long latency instructions compared to their SPEC counterparts as shown in Fig. 3(c)). (ii) However, the fetch stage, and the decode stage to some extent, are much more dominant in Android apps, compared to the SPEC ones (due to the drop in contribution from the Execute stage). As much as 40% of the time goes in the Fetch stage, while similar critical instructions in SPEC spend less than 5% of their time in this stage.

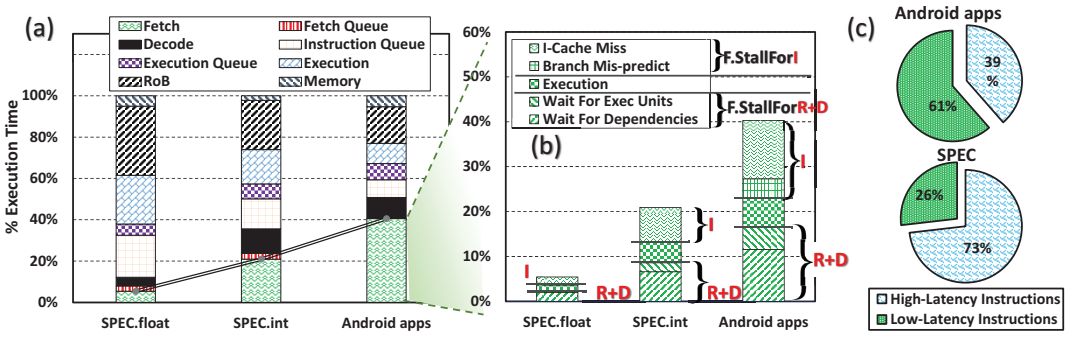


Fig. 3: (a) Fetch to Commit breakdown of high-fanout instructions in SPEC vs Android (b) In Android, Fetch is more bottlenecked due to both (i) stalling for instructions to be fetched (`F.StallForI`), and (ii) stalling for resources and dependencies (`F.StallForR+D`) to move the instructions down the pipeline. (c) Mobile apps have fewer high latency instructions compared to SPEC.

This shift in the profile from the rear to the front Fetch stage (consumes 40%) of the pipeline in Android, warrants us to take a closer look into this stage. Fig. 3(b) breaks down the Fetch execution time in these apps into two parts - `F.StallForI`, which is responsible for supplying the instruction stream into this stage, and the `F.StallForR+D` which pulls out the instruction from this stage for subsequent decoding. The former depends on the I-cache latency, miss costs, and branch misprediction costs, while the latter is largely determined by the back-pressure exerted by the subsequent pipeline stages (i.e., wait for decode to commit time for the prior instructions).

The relative contributions of the `F.StallForI` and `F.StallForR+D` (2:3) to the overall Fetch side overheads are quite comparable across the SPEC and Android apps. However, the actual values are quite different. While `F.StallForI` contributes to 3% of the overall execution in SPEC, Android apps execute from a much larger code base with a diverse set of libraries (>7k APIs [35]–[37]) with more frequent function calls, which causes i-cache stalls for 15% of the execution and branch prediction stalls for another 2% from the `F.StallForI`. At `F.StallForR+D`, SPEC apps execute many high-latency instructions that creates a back-pressure on the fetch stage by 3.6% (out of the 5.4% in SPEC.float) and 13% (out of 21% in SPEC.int). In Android apps, as shown in Fig. 3(c), majority of the high-fanout instructions are low-latency instructions, not imposing much back-pressure from the execute stage itself (6% out of 40%). Instead, the dependence resolutions between various instructions (as discussed in Fig. 1b) causes the most stall (11%) for the `F.StallForR+D` in these apps. Thus, any optimization for these critical instructions should try to reduce both `F.StallForI` and `F.StallForR+D` latencies, i.e., a simple i-cache/branch-prediction optimization, or a back-end optimization alone may not suffice as we will show later on.

Key Insights on Android Apps: (a) With high fanout, and thereby "critical" instructions occurring in close temporal proximity with one or more non-critical/low-fanout instructions in the dependence chain between them, we should consider optimizing groups/sequences of these instructions concurrently, rather than one at a time; (b) Fetch stage is much more important for these instructions, and optimizations for this stage are likely to yield more rewards than throwing more hardware for the conventionally bottlenecked execute-to-commit stages; (c) We need to accelerate not just the rate of bringing in the instructions to the Fetch stage, but also accelerate the rate of pushing out instructions into the rest of the pipeline.

III. CritICs: Critical Instruction Chains

Having identified the requirements, we now explore (i) how to identify these "critical" instructions occurring in a dependence chain/sequence in close temporal proximity, and (ii) how to optimize for these sequences to provide the minimal `F.StallForI` and `F.StallForR+D` latencies in the fetch stage with minimal hardware extensions.

A. Identifying CritICs

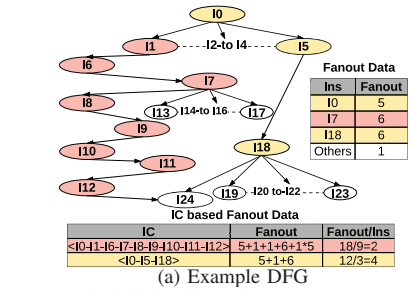
1) CritIC Sequences: As was shown in the example in Fig. 2a, identifying and optimizing for individual high fan-out, and thereby critical, instructions can only provide limited options. Instead, we need to look into the future, and find other possible future critical instructions which are in its forward dependence chain/graph. Consequently, the entire chain should be prioritized/optimized even if intermediate instructions in the forward dependence chain (such as I20 in the forward dependence chain of ⟨I10, I20, I22⟩) may not traditionally have been marked as critical because of their low fan-outs. Towards identifying such "*Critical Instruction Chains (CritIC)*", we first introduce the following metric and definitions.

Instruction Chain (IC): An instruction chain is any acyclic path of a Data Flow Graph (DFG) that is independently schedulable at that instant in the execution. In our previous example DFG of Fig. 2a:

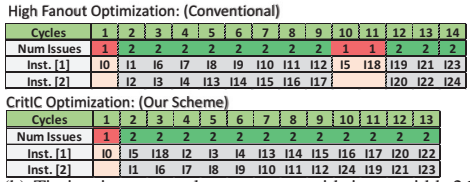
- The paths ⟨I0, I10, I20, I22⟩ and ⟨I0, I10, I11⟩ are independent of the other paths in the DFG. So, they are independently schedulable, and both qualify as ICs.
- The path ⟨I0, I1, I21⟩ does not qualify as an IC as it depends on another path, ⟨I0, I10, I11, I21⟩ and is thus not independently schedulable.
- Still, the sub-path ⟨I0, I1⟩ qualifies as an IC as it does not depend on any other paths of this DFG, i.e., any sub-path of an IC is also an IC.

An IC is thus a self-contained sequence of instructions, and is executable as an atomic entity (e.g., a macro instruction [38]–[44] consisting of several micro-instructions in the sequence) without any dependencies into its individual instructions. We will exploit this property later when optimizing critical ICs.

Crit: At any instant, a DFG has several individual ICs. The goal is then to find the right order for executing these ICs - to prioritize based on their relative criticalities. For example, in Fig. 4b, the execution at the top (high-fanout optimization) shows that prioritizing an IC with low-fanout instructions, ⟨I1, I6, I7, I8, I9, I10, I11, I12⟩ is inefficient - as observed in cycles 10, 11, the execution becomes serialized and there is no ILP for 2 cycles.



(a) Example DFG



(b) Timing in superscalar processor with issue width 2

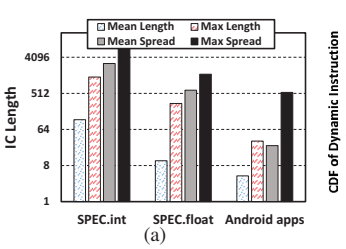
Fig. 4: Example: Need to optimize CritICs

However, identifying the relative criticalities of ICs is non-trivial, since each instruction in an IC can have a different fan-out/criticality. Simply adding up the fan-out of all its constituent instructions may not paint an accurate measure of an IC's criticality since there could be high variance amongst its instructions - a cumulatively high-fanout IC may have a very high fanout instruction at the beginning, with all subsequent instructions ending up with very low fanout, or vice-versa. While one could consider higher order representations for capturing such variances in future work, in this paper, we use a simple *average fanout per instruction of an IC* to capture the criticality of an IC. ***ICs whose average fanout per instruction exceed a certain threshold (e.g. 8) are marked as CritIC sequences in this work.*** Fig. 4 gives an example DFG, where a conventional instruction-level fanout based prioritization would give an execution as in the top part of (b) taking 14 cycles on a 2-way issue superscalar processor, while our CritIC approach would identify two ICs ⟨I1, I6, I7, I8, I9, I10, I11, I12⟩ and ⟨I0, I5, I18⟩ and prioritize the latter over the former because of its higher average fanout per instruction (4 vs. 2). This results in a schedule as in the lower part of (b), taking only 13 cycles.

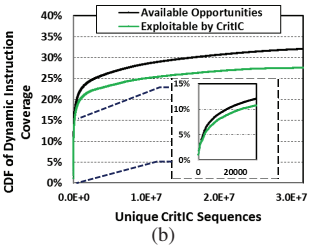
2) How to find them?: There are two broad strategies for identifying CritICs: (a) using hardware predictor tables as used in many prior works [4], [8], [12], [24] and/or (b) using software profile-driven compilation. As explained, we would like to minimize hardware requirements as much as possible, especially since mobile devices can become highly resource constrained. So we opt for the latter approach, which raises additional issues that we address as discussed below:

- *Ability to do this without User Intervention:* Unlike desktop environments where users may write their own apps, many of the mobile apps are published a priori (on the Play store, iTunes, etc.). It is not unreasonable for many of these popular apps to have undergone a profile-driven compiler optimization phase, which many of them already do (for quality, revisions, performance, bugs, etc. [45], [46]) before they get published. Our solution can be integrated into such phases for appropriate code generation.
- *Dealing with diverse inputs (user-interactivity):* Even if apps are available a priori, their execution can depend a lot on the input data - this is especially true for mobile apps which have high user interactivity. Conveniently, common cases of user inputs are readily provided for many of these apps in standard formats [47], [48], that we avail for our approach.
- *Ability to track long ICs and their spread:* Software based approaches are often criticized because of their restricted scope in analyzing large segments of code concurrently. This would pose a problem if the ICs were long and spread out considerably in the dynamic instruction stream. For instance, if we were to apply our approach for SPEC apps, Fig. 5a shows that we would need to track ICs of lengths up to 1.3K, which are spread over up to 6.3K instructions in the dynamic stream. On the other hand, in our favor, ICs for the mobile apps (as shown in the Fig. 5a), are at the maximum 20 instructions long, and are at most spread over 540 instructions to make them conducive to our approach.
- *Tractability of tracking all ICs:* Even when tracking 5 to 10 instruction long ICs, an app execution can generate a huge volume of profile data (100s of GB of `CritIC` sequences), with numerous sequences at any given instant. So, instead of tracking and optimizing for every possible `CritIC` sequence in an app, we track the top few CritIC sequences based on their coverage in the dynamic execution stream. This substantially reduces the profile size to a few kBs.

We have built this profile-driven compilation framework to



(a)



(b)

Fig. 5: (a)IC length and their corresponding spread in dynamic instruction execution in SPEC vs Android apps; (b)CDF of coverage by unique CritICs.

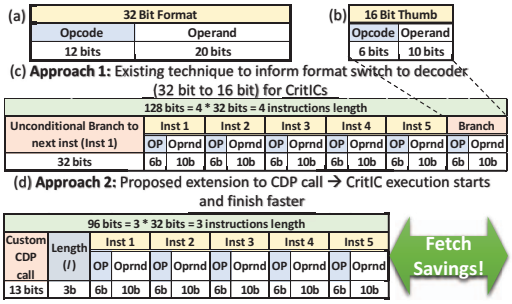


Fig. 6: Each `CritIC` instruction is transformed from the original 32-bit format to the 16-bit Thumb format of ARM ISA [51], [52].

automatically identify and optimize the CritIC sequences in a large number of Android apps. The app execution is profiled using AOSP emulation [22], [49] and GEM5 hardware simulator [23] to get the instruction stream from which we identify the CritIC sequences. The on-device Android Runtime Compiler (ART compiler) then generates optimized ARM binary using various compiler optimization passes [50]. After these passes, we have implemented an additional instrumentation pass in the compiler which visits every CritIC in the optimized DFG generated, to generate the optimization that is discussed next.

B. Optimizing CritIC Sequences

Our solution to optimizing these sequences is motivated by the two important observations: (i) `CritIC` sequence instructions spend nearly 40% of their execution in their fetch stage, with both `F.StallForI` and `F.StallForR+D` contributions becoming equally important; and (ii) Each `CritIC` sequence's instructions are "self-contained" and can execute in sequence without being influenced by any other sequence. Ideally, each `CritIC` sequence could, thus, be made a *macro-instruction* whose functionality is equivalent to executing each of its constituent instructions one after another. If our compiler could replace this entire sequence by the corresponding macro-instruction, we would avoid individual fetches for each of the constituents, and incur only 1 fetch operation - this would reduce the `F.StallForI` contribution. Further, by hoisting up this entire dependent chain of critical instructions into a single macro-instruction, we have reduced/eliminated any unnecessary gap between them, thus shortening the data flow from one to the other - this would reduce the `F.StallForR+D` contribution waiting for the later stages of the pipeline to flush out.

Creating Macro-Instructions: One obvious choice for implementing such macro-instructions is by extending the ISA with either (i) multiple mnemonics - one for each `CritIC` sequence, or (ii) having a new mnemonic with a passed argument that indexes a structure to find the `CritIC` sequence. In either case, the new macro-instruction has to know the exact sequence of micro-instructions that it needs to execute. This may be a reasonable option if the `CritIC` sequences are somewhat limited, i.e., there are a few common sequences which are widely prevalent across several apps as was the case in solutions such as [42], [43], [53], [54]. However, Fig. 5b shows that the number of unique `CritIC` sequences (opcode+operands of all constituent instructions) is large - even

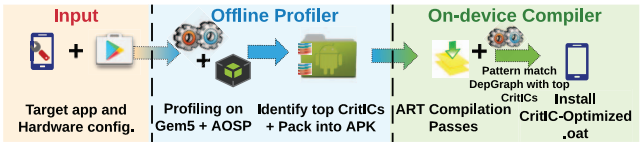


Fig. 7: Proposed Software Framework for our Methodology

each app can have 10^6 unique `CritIC` sequences - making it impossible to extend the ISA for this purpose, or building dedicated hardware for each unique `CritIC` sequence.

Exploiting ARM ISA: Instead, we need a mechanism for dynamically creating/mimic-ing such macro instructions based on the `CritICs` at hand, and we propose a novel way of achieving this in the ARM ISA. Fig. 6(a) shows the contemporary ARM ISA format [52] that uses 32 bits to represent an instruction - containing 12 to 20 bits for opcodes, 12 bits for representing 2 source and 1 destination operand registers. It also supports a concise format using 16-bits called "Thumb extension" (Fig. 6(b)). In this mode, the opcode is represented in 6 bits while the operands are represented in 3-4 bits each. The 16 bit format [52] is used in embedded controllers for optimizing binary size. The existing ARM decoders can decode any of these formats based on simple flags and pending queue structures [51].

We propose to represent each instruction of a `CritIC` sequence, that we would like to optimize, in the 16-bit format (Fig. 6(d)). Even though past studies [51], [52], [55] report that the 16-bit format produces $\approx 1.6\times$ more instructions to execute (and causes slowdown) because (i) it cannot have predicated executions, and (ii) it cuts the number of architected registers as operands from 16 to 11, we point out that the *16-bit format is very amenable for `CritIC` instructions* [1]. We illustrate this by plotting the CDF of coverage of the dynamic instruction stream by the instructions in all identified `CritIC` sequences of the original code (in 32 bit format) in Fig. 5b. In the same figure, we also plot the CDF of coverage by the `CritIC` instructions that can be represented in the 16-bit format without any change, i.e., they have neither predications nor use more than the allowed 11 registers. As we can see, there are very few `CritIC` instructions that cannot be represented (4.5% of the unique `CritIC` sequences), referred to as `CritIC.Ideal` in Sec. IV-E, which demonstrates the promise of our proposal.

Additionally, the ARM Decoder has to be informed of the instruction format, to switch back-and-forth between 32 and 16 bit representations. There are two possible ways to inform the decoder of the format switch: (i) in the current ARM hardware, this is done using explicit Branch instructions [52]. But, as we will show in Sec. IV-A, this incurs additional overheads especially for relatively short (< 10) CritIC instruction sequences; and (ii) our proposed alternative to extend an already existing instruction mnemonic to support CritIC thumb format switch in the decoder hardware (evaluated in Sec. IV-B).

C. Summarizing our Methodology

Fig. 7 summarizes the software framework for performing and implementing the `CritIC` optimizations:

[1] If any instruction of a `CritIC` sequence cannot be represented in the 16-bit format as is, then the entire sequence is left as is (in the original format) and is not optimized, i.e., all or nothing property (quantified in Fig. 5b).

- **Trace Collection:** We run the Android apps in QEMU [22] emulator with Android OS, where all the hardware components (CPU, GPU, touch, GPS, network, accelerometer, gyro, display, speakers, etc.) are modeled. We instrument its disassembler (with 1.6k lines of code or LOC) to output the trace of instructions executed and data accessed by the isolated process (app in consideration), for offline profiling.
- **Identifying CritICs:** This trace is used for detailed micro-architectural simulation in Gem5 [23], with modifications to identify critical instructions based on their fanouts across ROB entries (3.3k LOC). To get CritICs from the critical instructions, we implement additional tracking logic to dump all the independently schedulable ICs (whose lengths vary as discussed in Fig. 5) which results in 100s of GBs of ICs. These are processed offline with a distributed hash-table using Spark PairRDD [56] to sort and get the top CritICs (ICs with average fanout threshold > 8) with the most coverage (3.8k LOC). We fix 8 as the most beneficial average fanout threshold and also observe that other values result in slight performance degradations. The resulting CritICs is relatively concise ($\approx$10KB) to account for $\approx$30% of dynamic coverage.
- **Compilation:** Next, we modify the open-source ART compiler to add a final pass (CritIC instrumentation pass) that applies CritIC optimizations on the apk binary (.oat generation). Note that, the ART compiler already comes with different optimization passes such as `constant folding`, `dead code elimination`, etc., which work on DEX intermediate representation, as well as `load store elimination`, `register allocation`, etc., which work on the destination ARM assembly code before binary generation. Our CritIC pass works on ARM assembly code (similar to `instruction simplifier` pass) to take each CritIC (from the profile), checks whether each of its instructions are convertible into a 16-bit Thumb format, and if so, it lays down the entire `CritIC` sequence instructions one after another in this 16-bit format with appropriate two approaches explained next for switching the instruction format (1.8k LOC). Note that, other than hoisting and Thumb-converting the CritICs encountered, this pass does not affect the existing instruction scheduling.
- **Off the Shelf Apps:** Our framework can be readily applied to any off-the-shelf app (apk file) from the PlayStore [57]. Table II shows the ten mobile apps we use for evaluations. These apps belong to a diverse set of domains ranging from texting to gaming and video/audio streaming. These apps are also top rated and have millions of downloads in PlayStore.
- **Net Benefits:** We have roughly doubled the instruction fetch rate (halving `F.StallForI`) of the critical instruction sequences by switching formats, and reduced the `F.StallForR+D` delays by making this self-contained dependent chain contiguous in time. We will also demonstrate that our proposed solution has very little hardware overhead to interpret the format switch. In fact, our first approach can be readily done on current hardware, albeit with some inefficiencies as shown next. The second approach uses an existing mnemonic to switch format, which is a very small extension to the switch supported in existing ARM decoders.

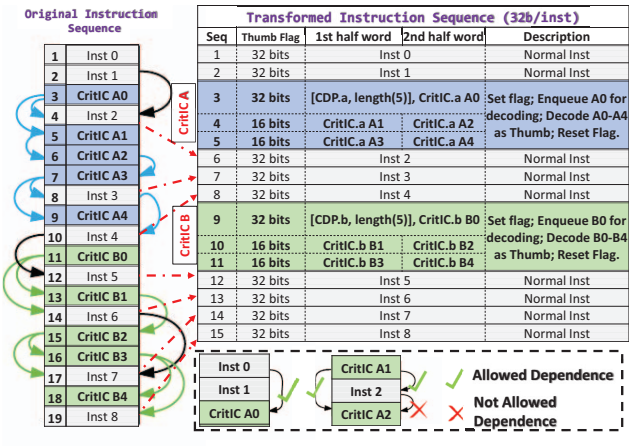


Fig. 9: Code Generation after `CritICs` have been identified. There are 2 `CritICs`, A and B, in this original instruction sequence.

IV. Evaluations

A. Switching Approach 1: On Actual Hardware

We use the conventional approach present in ARM decoders for switching between the two instruction formats, where two unconditional branch instructions are added at the beginning and end of each CritIC instruction sequence. The purpose of these branch instructions is to inform the decoder of the impending format switch and so both their target branch addresses are statically encoded to point to the subsequent instruction. As shown in Fig. 6, (i) the branch before the CritIC sequence, is in 32 bit format (that sets the Thumb flag at decode), and jumps to the first instruction of the CritIC; (ii) the subsequent 5 CritIC instructions are decoded in 16 bit Thumb format at the decoder; (iii) the branch after the CritIC sequence is also in 16 bit format, with its target set to the next instruction after the CritIC, that resets the format flag to 32-bit at the decoder. Note that the costs of these branches would mandate long CritIC sequences in order to amortize them.

We have implemented this on a Google Tablet hardware having 4 ARM cores and 2 GB LPDDR3 memory. Fig. 8 shows the gains on this hardware from our CritIC optimization for all 10 apps. Along with the actual speedup gains (of a measly 3% on the average), we also show the lost potential which we could have got if there were no branches before and after the CritICs for the format switches. We are getting only $\frac{1}{5}th$ of the possible gains since the CritIC sequences are not long enough (typically of length 5) to amortize the branch overheads. Motivated by this, we next propose an alternative, which does a very slight enhancement to the hardware, to address this problem to win back those gains.

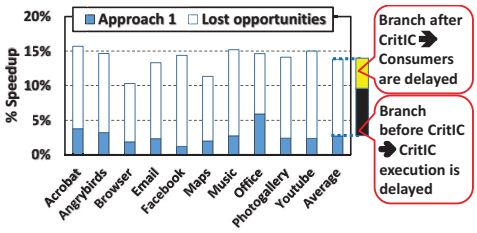


Fig. 8: Optimizing CritICs in existing hardware leaves 11% performance gap with the Ideal scenario.

B. Switching Approach 2: Extending Existing ARM Instruction

To avoid the aforementioned overheads, we propose to use an already existing instruction mnemonic, CDP (Co-processor

CPU	4 wide Fetch/Decode/Rename/ROB/Issue/Execute/Commit superscalar pipeline; 128 ROB entries, 4k Entry 2 level BPU [20], [59]
Memory System	2-way 32KB i-cache, 64KB d-cache, 2 cycle hit latency; 8-way 2MB L2 with CLPT prefetcher (1024×7bits entries) [18]; hit=10 cycles; 1 Ch;2 Ranks/Ch; 8 Banks per rank; open-page; Vdd = 1.2V; tCL,tRP,tRCD = 13, 13, 13 ns

TABLE I: Baseline Simulation configuration.

Domain	App	Activities Performed	Domain
Mobile	Acrobat	View, add comment	Document readers
	Angrybirds	1 Level of game	Physics games
	Browser	Search and load pages	Web interfaces
	Facebook	RT-texting	Instant messengers
	Email	Send,receive mail	Email clients
	Maps	Search directions	Navigation
	Music	2 minutes song	Music/audio players
	Office	Slide edit, present	Interactive displays
	PhotoGallery	Browse Images	Image browsing
	Youtube	HQ video stream	Video streaming
SPEC.int	bzip2, hmmer, libquantum, mcf, gcc, gobmk, sjeng, h264ref		
SPEC.float	sperand, namd, gromacs, calculix, lbm, milc, dealII, leslie3d		

TABLE II: Popular Mobile and SPEC apps used in evaluation.

Data Processing call), and the 3-bit argument with it to denote that the next $l+1$ instructions would be 16-bit format to inform the decoder accordingly. Fig. 9 illustrates this translation by our compiler pass for a `CritIC` sequence. In the first 32-bit word, the first half contains the CDP command, together with the l argument (Fig. 6(d)). The second half of this word contains the first instruction of the `CritIC` sequence in 16-bit format. The next $\lceil l/2 \rceil$ 32-bit words contain the next l instructions of the `CritIC` sequence in 16-bit format. Upon encountering the CDP command, the decoder puts the subsequent $l+1$ (1 coming in the latter half of the CDP word itself, and the other l coming from the remaining $\lceil l/2 \rceil$ words) `CritIC` instructions for 16-bit decoding. With the CDP argument having 3 bits, this allows us to translate up to $1+2^3 = 9$ `CritIC` instructions into the 16-bit format using a single CDP command. Note that we can also allow longer sequences by simply issuing more CDP commands subsequently, though we find that `CritIC` sequences up to 5 instructions suffice to provide the bulk of the savings (detailed in Sec. IV-H). After the last 16-bit instruction of this sequence passes through, the subsequent words get switched to the 32-bit decoding format. We also implemented and laid out the logic for the mode switch on CDP call on Synopsys Design Compiler(H-2013.03-SP5-2) [58] with 45 nm technology library and find that the extra logic only consumes $80\mu m^2$ area, dynamic and leakage power consumptions as $58\mu W$ and 414nW respectively. Although the timing for this logic is only 160ps, we conservatively assume a 1 cycle extra decoding stage delay when processing the CDP command.

Even though we have not cut the entire `CritIC` sequence down to one instruction fetch as in the above "macro-instruction" approach, our compiler-based ARM 16-bit translation roughly doubles the instruction fetch rate (halving `F.StallForI`) compared to the original alternative. Further, since these instructions are next to each other in the dynamic stream, the dataflow gap is reduced, thereby helping in the `F.StallForR+D` as well.

C. Simulation Results

We next describe the evaluation platform used for conducting our experiments on different design scenarios and conduct an in-depth evaluation of the proposed `CritIC` optimizations on performance and energy consumption.

Hardware: We evaluate the app executions using the hardware configuration of a Google Tablet in GEM5 [23]. As shown in Table I, this hardware consists of 4 CPUs, each with a 4-issue wide superscalar core, 32KB i-cache and a 64KB d-cache [60]. Further, we also simulate a detailed memory model for a 2GB LPDDR3 using DRAMSim2 [61], [62]. This setup enables us to execute apps in a cycle-level hardware simulation and obtain performance and power consumption for CPU, caches, and memory of the SoC.

App Execution: During the profiling phase (Sec. III-A2), these apps are emulated for an average of five minutes and execute, on average, around 100M instructions. This translates to ≈90 seconds of app execution time without the emulator overheads. For our evaluations, we pick 100 samples at random, each containing ≈500k contiguous instructions of app executions tallying to a total of ≈50 million instructions (same parts for all the optimizations evaluated).

D. Design Space

To quantify the performance effects of the proposed `CritIC` design on mobile apps, we evaluate three design choices, and compare them to the baseline configuration in Table I.

- **Hoist:** Since our solution employs two mechanisms - one hoisting all instructions of a `CritIC` sequence and another replacing them with 16-bit Thumb formats - we would like to study their effectiveness individually. Towards this, we implement a scheme which only does the former (i.e., identifies `CritIC` sequences, and hoists each sequences' instructions), but leaves them in 32-bit ARM format. We call this as Hoist in our evaluations.
- **CritIC:** This is our proposed `CritIC` design that aims to tackle the fetch side bottlenecks for high-fanout instructions as well as the `F.StallForR+D` bottlenecks by *hoisting/aggregating* the constituent instructions together and also translating these instructions to 16-bit Thumb format.
- **CritIC.Ideal:** As was noted earlier in Fig. 5b, we choose to leverage only a subset of the total number of `CritIC` sequences - (i) those that are at most length 5, and (ii) those whose instructions can be translated directly to the 16-bit Thumb format. In order to find out the lost opportunity, we also evaluate a scheme called `CritIC.Ideal` which hypothetically aggregates and Thumb-translates for all `CritIC` instructions (i.e., the black CDF of Fig. 5b).

E. Performance Results

Fig. 10a plots the CPU execution speedup of each app for the three scenarios discussed above to study the individual as well as combined effects of the two components of CritIC optimizations. We discuss app level speedups of each of these optimizations *normalized* with respect to the baseline design. When we consider the individual optimizations evaluated in Fig. 10a, we see that the `CritIC` optimizations consistently perform well in all apps with 9% (Music) to 15% (Acrobat) speedup. However, `Hoist` (which only targets StallforRD) by itself, only gives marginal improvements (average gain of 2.5%) compared to CritIC which combines both `F.StallForI` and `F.StallForR+D` optimizations, suggesting that just moving

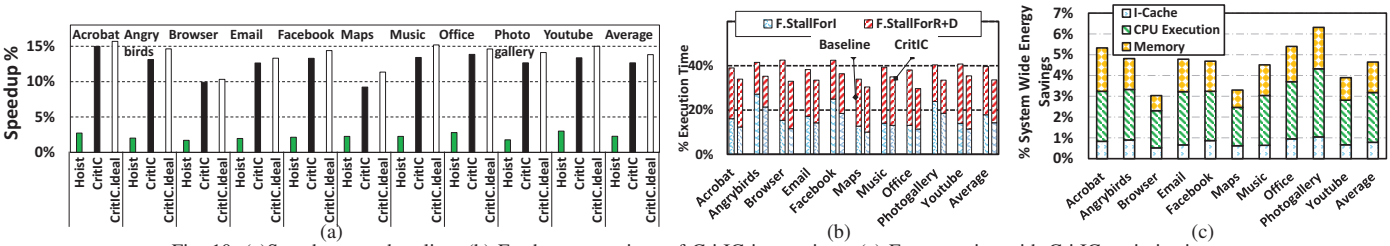


Fig. 10: (a)Speedup over baseline; (b) Fetch stage savings of CritIC instructions; (c) Energy gains with CritIC optimization

instructions does not suffice. Since this scheme only reduces the dataflow gap across critical instructions, without boosting the fetch efficiency, the impact of just a `F.StallForR+D` optimization is not felt across these apps, reiterating the need for fetch side improvements. Of the apps, Maps and Youtube are more bottlenecked in the `F.StallForR+D` (26.7% in Youtube in baseline of Fig. 10b) and this in turn translates to the most benefits when it comes to optimizations for `F.StallForR+D` (3.1%). All the other apps have even less improvements from hoisting the CritIC instructions, with Browser and Photogallery showing the least benefits of 1.7%. `CritIC`, which implements both 16-bit conversions to boost the fetch bandwidth, as well as the `Hoist` improvements, gives 12.6% speedup improvements on the average. In fact, we see that the differences between `CritIC` and `CritIC.Ideal`, to be quite small (e.g. only 1% gap in Acrobat, Browser and Office). Limiting ourselves to CritIC lengths of 5 or to those that can be directly translated to 16-bit Thumb format, does not seem to hurt. This is because, a majority of CritIC instructions are amenable to 16-bit Thumb representation, leaving $<$1% room for any further improvement on the average. As discussed in Sec. III, the volume of CritIC instructions representable with the 16-bit format is within 5% of the entire CritIC instruction volume. We note that the average 12.6% speedup with CritIC significantly outperforms the previously proposed single instruction criticality optimizations - load prefetching and ALU prioritization - for which we showed speedups of 0.7% and 4.1% respectively.

F. System-Wide Energy Gains

The effect of our `CritIC` optimizations in terms of the energy gains from various components of the mobile SoC is plotted in Fig. 10c. Recall that `CritIC` optimizations decrease the number of accesses to the i-cache by 40% (Fig. 6) for each IC execution by representing 5 $\times$ 32-bit instructions as 3 $\times$ 32-bit instructions. This translates to energy gains from i-cache by 0.8% for the whole SoC. The CPU speedup discussed above also results in additional energy gains for both CPU and memory. On an average, CPU contributes to 2.2% of the energy savings and the memory side of the execution contributes an additional 1.5%. Overall, we observe 4.6% energy saving for the *whole system* on the average, with the maximum energy savings of 6.3% (in Photogallery). Specifically, the CPU execution alone (excluding peripherals, ASIC accelerators, etc.) realizes an average energy saving of 15%.

GComparing with Conventional Hardware Fetch Optimizations

One may note that numerous prior hardware enhancements proposed to address the Fetch stage problems, including larger

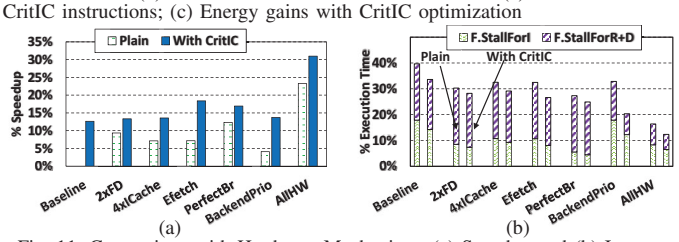


Fig. 11: Comparison with Hardware Mechanisms (a) Speedup and (b) Impact on `F.StallForI`and `F.StallForR+D`.

and more intelligently managed i-caches [63]–[65], better branch predictors [66]–[69], and/or instruction prefetchers [70]–[74]. While adding sophisticated hardware for high end CPUs may be acceptable, the resource constraints of mobile platforms may not warrant such sophisticated hardware. Still, we have implemented a number of hardware solutions for addressing the Fetch bottleneck (described below), and compared them to the speedup obtained with our software-only solution – CritIC:

- **2$\times$FD:** Since CritIC uses a 16-bit format to put 2 instructions into each fetched word (selectively doubling fetch bandwidth for critical instructions), we consider a hypothetical hardware where the Fetch and Decode stage bandwidths are doubled (for all instructions - not just critical ones), with no change to other stages. In this scheme (2$\times$FD), we simulate a hardware with half the i-cache latency and double the resources (hardware units/queues) in the fetch and decode stages.
- **4$\times$i-cache:** Though unreasonable, we compare with a hardware that has 4$\times$ the i-cache capacity (128KB vs. 32KB) to reduce instruction misses.
- **EFetch [71]:** We implemented a recently proposed instruction prefetcher [71] that is specifically useful for user-event driven applications, as in our mobile apps. This prefetcher [71] tracks history of user-event call stack, and uses it to predict the next functions and prefetch its instructions. It needs a 39KB lookup table for maintaining the call stacks.
- **PerfectBr:** This is a hypothetical system where we assume there is no branch misprediction in the entire execution.

Since CritIC addresses both (i) `F.StallForI` which the above 3 address; and (ii) `F.StallForR+D`, which is somewhat addressed by prior criticality optimizations such as [5], [6], [14], [16], [25], [31], [75]–[77], which prioritize the back-end resources for those instructions, we additionally consider the following configurations:

- **BackendPrio [33]:** This platform implements the prioritization hardware for the back-end resources proposed in [33], using the tracking hardware proposed in [32], which requires 1.5KB SRAM for maintaining the tokens.
- **AllHW:** This consists of hardwares for both front and

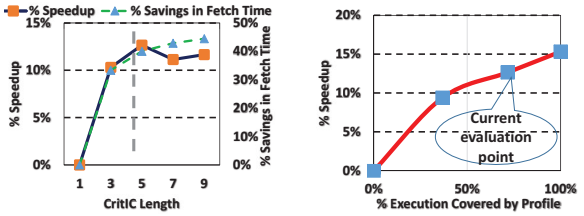


Fig. 12: Sensitivity Analysis: (a) Fetch savings and speedup w.r.t CritIC length; and (b) Speedup w.r.t CritIC Profile Coverage.

backends, i.e., 4×i-cache+EFetch+PerfectBr+BackendPrio.

- **With CritIC:** In addition to comparing with vanilla CritIC, which has no additional hardware needs, we also study CritIC in combinations with every above hardware mechanisms.

Results: We observe in Fig. 11a that previously proposed hardware mechanisms yield ≈4% to 12% speedup. However, it is important to optimize for both `F.StallForI` and `F.StallForR+D`. These hardware mechanisms only benefit one of these two stalls (Fig. 11b). For example, 2×FD, 4× larger i-cache and EFetch lower miss penalties to reduce the `F.StallForI` by ≈7%, while PerfectBr completely eliminates branch penalties to reduce fetch stalls by 12%. These mechanisms have no effect on `F.StallForR+D`. Similarly, BackendPrio only addresses the `F.StallForR+D` problem, reducing it by 3% and does not tackle the `F.StallForI`.

While one could throw all this hardware to tackle both these stalls, as in AllHW, to get the overall speedup benefits of 23.2%, such extensive hardware may be unacceptable for a mobile platform. CritIC, by itself, which does need any additional hardware, does significantly better than each of these individual hardware mechanisms. If future mobile platforms are to incorporate one or more of these `F.StallForI` and `F.StallForR+D` hardware mechanisms, our results in Fig. 11a show that CritIC can synergistically boost the benefits further. In fact, even with a system that incorporates all of the above hardware (AllHW) which gives a speedup of 23.2%, can be boosted to give a speedup of 31% with CritIC on top.

H. Sensitivity to CritIC length

The speedup and energy gains reported above are for a small CritIC size of 5 instructions. We next investigate the impact of CritIC length on application performance.

Even though `CritIC.Ideal` showed not much difference compared to the realistic `CritIC` (which uses lengths of up to 5 instructions), it is interesting to see which CritIC length gives the most rewards individually, i.e., not just all CritICs up to length n, but for each individual n. Note that as n increases, we are saving more on the fetch costs - both `F.StallForI` and `F.StallForR+D` latencies. However, the probability of finding a CritIC of exactly length n, where all its n instructions can be directly translated to the 16-bit Thumb format, decreases as n increases. To study these trade-offs, in Fig. 12a we study the impact of a given n (x-axis) on the fetch cost savings (right y-axis) and the consequent speedup (left y-axis). As expected, fetch costs keep dropping with larger n, though with diminishing returns. The speedup increases up to a point ($n = 5$), beyond which it starts dropping since the probability of finding such sequences diminishes. In fact, we observe a drop in coverage of CritICs executed from 16% to 15% as we move for a longer CritIC.

I. Sensitivity to Profiling

Since our technique uses offline profiling to identify and modify critical chains, we also study the sensitivity of results to the extent of profiling, i.e. the percentage of the app execution that is profiled. Fig. 12b shows the speedup (y-axis) as a function of the percentage of the execution that is profiled (x-axis), averaged across all apps. The results presented so far use profiling that covers 72% of the execution. While a lower coverage does reduce the speedup obtained, we see that even when only a third of the execution is profiled and transformed into CritIC thumb sequences, we still get 10% speedup across these apps. If we further the profiling, and transform the entire application, we can get up to 15% speedup on the average.

V. Why even bother with criticality?

While we have proposed the use of Thumb 16-bit format to nearly double the fetch bandwidth of the `CritIC` instructions, one may use this approach opportunistically for all instructions amenable to such modification in the instruction stream. If so, one could question why we bothered to identify `CritICs` in the first place. To justify the need, in Fig. 13a, we plot the speedup obtained with the following schemes:

- **OPP16:** In this approach, we opportunistically convert any amenable sequence of consecutive dynamic instructions (sequence has to be of at least length 3) to the 16-bit Thumb format, regardless of whether they are critical or not. Note that if there is an instruction which is not amenable to such format conversion between two other instructions which are amenable, OPP16 will NOT move the instructions around for the conversion. Also, as explained earlier, if the dynamic sequence exceeds 9 contiguous instructions that can be converted, we use another CDP instruction to accommodate longer sequences for such conversion.
- **Compress:** This is a state-of-the-art thumb compression technique, implementing the *Fine-Grained Thumb Conversion* heuristic from [78], that first converts a whole function to Thumb, then replaces frequently occurring "slower thumb instructions" back to 32 bit ARM instructions.
- **CritIC:** This implements our `CritIC` mechanism described earlier, moving/hoisting identified `CritIC` sequence instructions and converting them to 16-bit format as long as they are amenable to such conversion and they are of length ≤ 5.
- **OPP16+CritIC:** We combine `CritIC` (for `CritIC` sequence instructions) and OPP16 (for others) in this approach.

As seen, just opportunistically leveraging the 16-bit Thumb format (in OPP16) only provides 6% benefit on the average

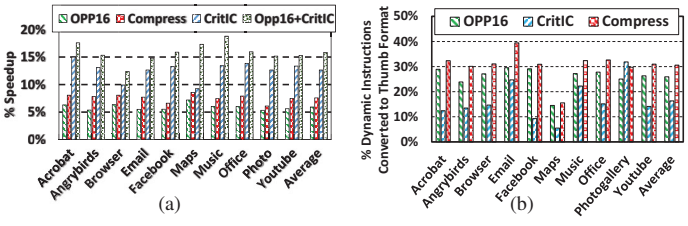


Fig. 13: Opportunistically transforming to 16-bit Thumb format. (a) Speedup and (b) Percentage of Dynamic Instructions converted to 16-bit format.

over the baseline. Even smartly employing the Thumb format (Compress), as in [78], only yields a 8% speedup. Since both OPP16 and Compress are agnostic to critical instruction chains, they can only save on fetch costs (`F.StallForI`) whenever possible without hoisting the dependent instructions in the chain. Hence, both these techniques provide less than 40% of the benefits provided by our `CritIC` optimization, even though as shown in Fig. 13b, `CritIC` converts around 37% and 50% fewer instructions in the dynamic stream to the 16-bit format compared to OPP16 and Compress respectively. This clearly points out the need to identify the critical instruction sequences for such optimization, instead of blindly doing this for all instructions. In fact, nothing precludes adding on the optimization for other instructions on top of `CritIC`, as is shown for OPP16+CritIC schemes, furthering the speedup by 25% over doing `CritIC` alone.

VI. Related Work

Criticality: Instruction criticality has been shown to be an important criterion in selectively optimizing the instruction stream. Prior work has revolved around both (i) identifying critical instructions [4], [9], [12], [24]–[26] using metrics such as fanout, tautness, execution latencies, slack, and execution graph representations, as well as (ii) optimizing for those identified using techniques such as critical load optimizations [9], [11], [12], [18], [79] or even backend optimizations for critical instructions such as [5], [6], [14], [16], [24], [25], [31], [75]–[77]. While one can potentially employ these optimizations for mobile apps, as we showed (in Fig. 1b), mobile apps have close data-dependent, clustered occurrences of critical instructions, requiring their ensemble optimization rather than their consideration individually.

Optimizing Instruction Chains/Ensembles: There are prior works, specifically for high-end processors, in identifying and extracting dependence chains [80]–[82]. However, such techniques require fairly extensive hardware to identify these chains, and optimizing for them, e.g. techniques such as [18], [76], [77] require 16KB SRAM, and [79] incurs 22% additional power, making them less suitable for resource-constrained mobile SoCs. In contrast, our solution is an entirely software approach for identifying dependence chains, and a software approach in optimizing for them by intelligently employing the ARM 16-bit thumb compression [19]–[21], [51] mechanism.

Front-end Optimizations for Mobile Platforms: There has been significant recent interest to optimize mobile CPU execution [83]–[89]. Some of these optimizations target specific domains (e.g. web-browsers [90]–[93]), while others address overall efficiency [37], [94]–[96]. Unlike our approach, many of these optimizations either provision more CPU hardware [90], [95], [96], or optimize for only specific app domains [90]–[92]. This paper is amongst the first to show that mobile apps are bottlenecked in the Fetch stage of the pipeline, suggesting that there can be considerable rewards in targeting this stage. Fetch stage bottlenecks have been extensively addressed in high end processors through numerous techniques - smart i-cache management (e.g. [63]–[65], [97]–[99]) prefetching (e.g. [70]–[74]), branch prediction (e.g [66]–[69]), instruction compression [100] SIMD [38], [39], VLIW [40], vector processing [41], etc. However, many of these require extensive hardware that mobile platforms may not be conducive for. As we showed, our software solution employs a simple trick of hoisting and Thumb conversion on critical instructions to extract the same performance that many of these high-end hardware mechanisms provide. Further, as mobile processors evolve to incorporate more hardware for optimizing the fetch stage, as shown, our CritIC software approach can synergistically integrate with them to significantly boost the improvements. While similar in spirit to some of the prior work on instruction stream compression [101]–[103], we quantitatively showed the need to identify critical chains and hoisting the instructions selectively before doing the compression.

Software Profiling for Mobile Platforms: A number of software profiling frameworks have been proposed [35], [104]–[107] - studying library usage [35], [106], app-market level changes to the source/advertisement models, [104], [105], dynamic instrumentation mechanisms [107], developer side debugging/optimizations [108], [109] etc. Some of these tools can also be extended for the profiling and compilation phases described in this work. We have built on top of the AOSP emulation [22], [49] and Gem5 hardware simulator [23] for profiling, and ART compiler for code transformation.

VII. Conclusion

This paper targets to enhance the performance a growing class of applications - mobile apps – that are more prevalent and user driven than traditional server/scientific workloads. In this context, we show that mobile apps have unique characteristics such as high volume of critical instructions occurring as short sequences of dependent instructions that makes them less attractive for exploiting well-known criticality-based optimization techniques. We instead introduce the concept of `CritICs` as a granularity for tracking and exploiting criticality in these apps. We present a novel profiler-driven approach to identify these `CritICs`, and hoist and aggregate them by exploiting existing ARM ISA's Thumb instruction format in a compiler pass to boost the front-end fetch bandwidth. The end-to-end design starting from application profiling, identification of `CritICs`, hoisting those instructions and transformation them to the 16-bit Thumb format has been evaluated for a Google Tablet using the GEM5 simulator to estimate the performance and energy benefits. Evaluations with ten popular mobile apps indicate that the proposed solution results in an average 12.6% speedup and 4.6% reduction in system-wide energy consumption compared to the baseline design, requiring little to no hardware support. The proposed technique can also be synergistically integrated with other optimizations such as hardware prefetching, or even opportunistically converting as many instructions as possible to the Thumb format, to further the benefits.

Acknowledgment

This work has been supported in part by NSF grants 1439021, 1629915, 1526750, 1629129, 1763681, 1409095, 1317560, 1320478, 182293, 162651 and 1439057, and a DARPA/SRC JUMP grant. We would also like to thank Jack Sampson for his feedback on this paper.

REFERENCES

[1] Statista.com, "Number of Smartphone Users WorldWide," https://goo.gl/PCGQPA, 2015.

[2] SmartInsighs.com, "Mobile Marketing Statistics," https://goo.gl/vNpuEr, 2015.

[3] TechCrunch.com, "1B Smartphone Users Globally by 2020 Overtaking Basic Fixed Phone Subscriptions," https://goo.gl/aoSN5q, 2015.

[4] B. Fields, R. Bodik, and M. D. Hill, "Slack: Maximizing Performance Under Technological Constraints," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2002.

[5] R. Nagarajan, X. Chen, R. G. McDonald, D. Burger, and S. W. Keckler, "Critical Path Analysis of the TRIPS Architecture," in *2006 IEEE International Symposium on Performance Analysis of Systems and Software*, 2006.

[6] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg, "A Large, Fast Instruction Window for Tolerating Cache Misses," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2002.

[7] H. Zhang, W. Han, F. Li, S. He, Y. Cheng, H. An, and Z. Chen, "A Criticality-Aware DVFS Runtime Utility for Optimizing Power Efficiency of Multithreaded Applications," in *IEEE International Parallel Distributed Processing Symposium Workshops*, 2014.

[8] B. R. Fisk and R. I. Bahar, "The Non-critical Buffer: Using Load Latency Tolerance to Improve Data Cache Efficiency," in *Proceedings 1999 IEEE International Conference on Computer Design: VLSI in Computers and Processors (Cat. No.99CB37040)*, 1999.

[9] S. T. Srinivasan, R. D.-c. Ju, A. R. Lebeck, and C. Wilkerson, "Locality vs. Criticality," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2001.

[10] Prasanna Venkatesh Rengasamy and Madhu Mutyam, "Using Packet Information For Efficient Communication in NoCs," Sep 2014.

[11] S. Ghose, H. Lee, and J. F. Martínez, "Improving Memory Scheduling via Processor-side Load Criticality Information," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2013.

[12] S. T. Srinivasan and A. R. Lebeck, "Load Latency Tolerance in Dynamically Scheduled Processors," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1998.

[13] Jayesh Gaur and Prasanna Rengasamy and Pradeep Ramachandran and Sreenivas Subramoney, "Instruction and Logic for Managing Cumulative System Bandwidth through Dynamic Request Partitioning," Jun 2016.

[14] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt, "Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2003.

[15] S. T. Srinivasan, R. Rajwar, H. Akkary, A. Gandhi, and M. Upton, "Continual Flow Pipelines," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2004.

[16] H. Zhou, "Dual-core Execution: Building a Highly Scalable Single-thread Instruction Window," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2005.

[17] Prasanna Venkatesh Rengasamy and Anand Sivasubramaniam and Mahmut Taylan Kandemir and Chita R Das, "Exploiting staleness for approximating loads on CMPs," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Oct 2015.

[18] S. Subramaniam, A. Bracy, H. Wang, and G. H. Loh, "Criticality-based Optimizations for Efficient Load Processing," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2009.

[19] arm.com, "Cortex-A75," "https://developer.arm.com/products/processors/cortex-a/cortex-a75", 2018, [Online].

[20] arm.com, "Cortex-A55," "https://developer.arm.com/products/processors/cortex-a/cortex-a55", 2018, [Online].

[21] arm.com, "Cortex-A35," "https://developer.arm.com/products/processors/cortex-a/cortex-a35", 2018, [Online].

[22] QEMU Project, "The Fast! processor emulator," http://www.qemu-project.org/, 2017.

[23] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 Simulator," *SIGARCH Comput. Archit. News*, 2011.

[24] E. S. Tune, D. M. Tullsen, and B. Calder, "Quantifying Instruction Criticality," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2002.

[25] T. Li, A. R. Lebeck, and D. J. Sorin, "Quantifying Instruction Criticality for Shared Memory Multiprocessors," in *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, 2003.

[26] E. Tune, "Critical-path Aware Processor Architectures," Ph.D. dissertation, 2004.

[27] D. R. Kerns and S. J. Eggers, "Balanced Scheduling: Instruction Scheduling when Memory Latency is Uncertain," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1993.

[28] E. Schnarr and J. R. Larus, "Instruction Scheduling and Executable Editing," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1996.

[29] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen, "Dynamic Speculative Precomputation," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2001.

[30] F. Gabbay and A. Mendelson, "The Effect of Instruction Fetch Bandwidth on Value Prediction," *SIGARCH Comput. Archit. News*, 1998.

[31] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A High-Performance Sparc CMT Processor," *IEEE Micro*, 2009.

[32] B. Fields, S. Rubin, and R. Bodik, "Focusing Processor Policies via Critical-path Prediction," in *Proceedings 28th Annual International Symposium on Computer Architecture*, 2001.

[33] R. Pyreddy and G. Tyson, "Evaluating design tradeoffs in dual speed pipelines," in *Workshop on Complexity-Effective Design in conjunction with ISCA*, 2001.

[34] E. Tune, D. Liang, D. M. Tullsen, and B. Calder, "Dynamic Prediction of Critical Path Instructions," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2001.

[35] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh, "POSIX Abstractions in Modern Operating Systems: The Old, the New, and the Missing," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[36] Google, "Android Developer Reference Doc," https://developer.android.com/reference/packages.html, 2017.

[37] Prasanna Venkatesh Rengasamy and Haibo Zhang and Nachiappan Chidhambaram Nachiappan and Shulin Zhao and Anand Sivasubramaniam and Mahmut Kandemir and Chita R Das , "Characterizing Diverse Handheld Apps for Customized Hardware Acceleration ," in *In Proceedings of IEEE International Symposium on Workload Characterization*, Oct 2017.

[38] S. Thakkur and T. Huff, "Internet Streaming SIMD Extensions," *Computer*, 1999.

[39] N. Firasta, M. Buxton, P. Jinbo, K. Nasri, and S. Kuo, "Intel AVX: New Frontiers in Performance Improvements and Energy Efficiency," *Intel white paper*, 2008.

[40] P. Faraboschi, G. Brown, J. A. Fisher, G. Desoll, and F. M. O. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2000.

[41] S. R. Cray, "Computer Vector Register Processing," 1976, US Patent 4,128,880.

[42] C. IP, "Tensilica Custimizable Processor and DSP IP," https://ip.cadence.com/ipportfolio/tensilica-ip, 2017.

[43] C. Gonzlez-lvarez, J. B. Sartor, C. lvarez, D. Jimnez-Gonzlez, and L. Eeckhout, "Automatic Design of Domain-specific Instructions for Low-power Processors," in *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 2015.

[44] C. Celio, P. Dabbelt, D. A. Patterson, and K. Asanović, "The Renewed Case for the Reduced Instruction Set Computer: Avoiding ISA Bloat with Macro-Op Fusion for RISC-V," *arXiv preprint arXiv:1607.02318*, 2016.

[45] Apple, "App Review," https://developer.apple.com/app-store/review/, 2017.

[46] Android, "App Quality," https://developer.android.com/develop/quality-guidelines/core-app-quality.html, 2017.

[47] Android, "Monkeyrunner," https://developer.android.com/studio/test/monkeyrunner/index.html, 2017.

[48] S. J. Patel and S. S. Lumetta, "rePLay: A Hardware Framework for Dynamic Optimization," *IEEE Transactions on Computers*, 2001.

[49] Google, "Android Open Source Project (AOSP)," https://github.com/android, 2017.

[50] Android, "ART Compiler Optimizing," http://android.googlesource.com/platform/art/+/master/compiler/optimizing/, 2017.

[51] X. H. Xu, C. T. Clarke, and S. R. Jones, "High Performance Code Compression Architecture for the Embedded ARM/THUMB Processor," in *Proceedings of the 1st Conference on Computing Frontiers*, 2004.

[52] ARM, "ARM Architecture Reference Manual," https://www.arm.com/products/processors/technologies/biglittleprocessing.php, 2017.

[53] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled Execution of Recurring Traces for Energy-efficient General Purpose Processing," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2011.

[54] F. Sun, S. Ravi, A. Raghunathan, and N. K. Jha, "A Scalable Application-specific Processor Synthesis Methodology," in *ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No.03CH37486)*, 2003.

[55] K. D. Cooper and N. McIntosh, "Enhanced Code Compression for Embedded RISC Processors," *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, 1999.

[56] Apache, "spark.api.java.JavaPairRDD," https://spark.apache.org/docs/0.7.2/api/core/spark/api/java/JavaPairRDD.html, 2016.

[57] Google, "Google Play," https://play.google.com/store?hl=en, 2016.

[58] Synopsys, "DC Ultra," https://www.synopsys.com/implementation-and-signoff/rtl-synthesis-test/dc-ultra.html, 2013.

[59] AnandTech, "ARM A53/A57/T760 investigated - Samsung Galaxy Note 4 Exynos Review," "https://developer.arm.com/products/processors/cortex-a/cortex-a75", 2015, [Online].

[60] A. Inc, "Nexus 7 Tablet Specifications," https://goo.gl/aPRBuw, 2013.

[61] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "DRAMSim2: A Cycle Accurate Memory System Simulator," *IEEE Computer Architecture Letters*, 2011.

[62] MICRON, "Production Data Sheet: 8Gb, 16Gb: 253-Ball, Dual-Channel 2C0F Mobile LPDDR3 SDRAM (pdf)," "https://goo.gl/JELDcz", 2016, [Online; accessed March-29-2017].

[63] J. Kim, E. Teran, P. V. Gratz, D. A. Jiménez, S. H. Pugsley, and C. Wilkerson, "Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy," *SIGOPS Oper. Syst. Rev.*, 2017.

[64] W. A. Wong and J. L. Baer, "Modified LRU policies for improving second-level cache behavior," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2000.

[65] H. S. Stone, J. Turek, and J. L. Wolf, "Optimal partitioning of cache memory," *IEEE Transactions on Computers*, 1992.

[66] A. Seznec, "A New Case for the TAGE Branch Predictor," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2011.

[67] H. Akkary, R. Rajwar, and S. T. Srinivasan, "Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2003.

[68] T.-Y. Yeh and Y. N. Patt, "Two-level Adaptive Training Branch Prediction," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1991.

[69] C. Zhou, L. Huang, T. Zhang, Y. Wang, C. Zhang, and Q. Dou, "Effective Optimization of Branch Predictors through Lightweight Simulation," in *IEEE International Conference on Computer Design (ICCD)*, 2017.

[70] A. Jain and C. Lin, "Rethinking Beladyś Algorithm to Accommodate Prefetching," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2018.

[71] G. Chadha, S. Mahlke, and S. Narayanasamy, "Efetch: Optimizing instruction fetch for event-driven web applications," in *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Aug 2014.

[72] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 1990.

[73] X. Yu, S. K. Haider, L. Ren, C. Fletcher, A. Kwon, M. van Dijk, and S. Devadas, "Proram: Dynamic prefetcher for oblivious ram," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[74] P. Michaud, "Best-offset hardware prefetching," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2016.

[75] J. F. Martínez, J. Renau, M. C. Huang, M. Prvulovic, and J. Torrellas, "Cherry: Checkpointed Early Resource Recycling in Out-of-order Microprocessors," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2002.

[76] A. Perais and A. Seznec, "BeBoP: A Cost Effective Predictor Infrastructure for Superscalar Value Prediction," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2015.

[77] A. Perais and A. Seznec, "EOLE: Paving the Way for an Effective Implementation of Value Prediction," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2014.

[78] A. Krishnaswamy and R. Gupta, "Profile Guided Selection of ARM and Thumb Instructions," in *Proceedings of the Joint Conference on Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, ser. LCTES/SCOPES, 2002.

[79] T. E. Carlson, W. Heirman, O. Allam, S. Kaxiras, and L. Eeckhout, "The Load Slice Core microarchitecture," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[80] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam, "SlicK: Slice-based Locality Exploitation for Efficient Redundant Multithreading," *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, vol. 40, Oct. 2006.

[81] M. D. Brown, J. Stark, and Y. N. Patt, "Select-free instruction scheduling logic," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 2001.

[82] F. Tip, "A survey of program slicing techniques." Amsterdam, The Netherlands, The Netherlands, Tech. Rep., 1994.

[83] M. Halpern, Y. Zhu, and V. J. Reddi, "Mobile CPU's Rise to Power: Quantifying the Impact of Generational Mobile CPU Design Trends on Performance, Energy, and User Satisfaction," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2016.

[84] N. C. Nachiappan, H. Zhang, J. Ryoo, N. Soundararajan, A. Sivasubramaniam, M. T. Kandemir, R. Iyer, and C. R. Das, "Vip: Virtualizing ip chains on handheld platforms," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[85] A. Boroumand, S. Ghose, Y. Kim, R. Ausavarungnirun, E. Shiu, R. Thakur, D. Kim, A. Kuusela, A. Knies, P. Ranganathan, and O. Mutlu, "Google Workloads for Consumer Devices: Mitigating Data Movement Bottlenecks," in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[86] W. Wang, P.-C. Yew, A. Zhai, S. McCamant, Y. Wu, and J. Bobba, "Enabling Cross-ISA Offloading for COTS Binaries," in *Proceedings of the ACM Annual International Conference on Mobile Systems, Applications, and Services (MOBISYS)*, 2017.

[87] J. Wan, R. Wang, H. Lv, L. Zhang, W. Wang, C. Gu, Q. Zheng, and W. Gao, "AVS Video Decoding Acceleration on ARM Cortex-A with NEON," in *Proceedings of International Conference on Signal Processing, Communication and Computing (ICSPCC)*, 2012.

[88] Haibo Zhang and Prasanna Venkatesh Rengasamy and Shulin Zhao and Nachiappan Chidambaram Nachiappan and Anand Sivasubramaniam and Mahmut Kandemir and Ravi Iyer and Chita R. Das, "Race-To-Sleep + Content Caching + Display Caching: A Recipe for Energy-efficient Video Streaming on Handhelds ," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, Oct 2017.

[89] Haibo Zhang and Prasanna Venkatesh Rengasamy and Nachiappan Chidambaram Nachiappan and Shulin Zhao and Anand Sivasubramaniam and Mahmut Kandemir and Chita R. Das, "FLOSS: FLOw Sensitive Scheduling on Mobile Platforms," in *Proceedings of the Design and Automation Conference (DAC)*, 2018.

[90] Y. Zhu and v. J. Reddi, "WebCore: Architectural Support for Mobileweb Browsing," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2014.

[91] Y. Zhu and V. J. Reddi, "GreenWeb: Language Extensions for Energy-efficient Mobile Web Computing," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2016.

[92] G. Chadha, S. Mahlke, and S. Narayanasamy, "Accelerating Asynchronous Programs Through Event Sneak Peek," in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, 2015.

[93] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn, "Race Detection for Event-driven Mobile Applications," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2014.
[94] K. Yan and X. Fu, "Energy-efficient Cache Design in Emerging Mobile Platforms: The Implications and Optimizations," in *Proceedings of the Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2015.
[95] D. Boggs, G. Brown, B. Rozas, N. Tuck, and K. S. Venkatraman, "NVIDIA's Denver processor," *HOTChips*, 2011.
[96] ARM, "ARM big.LITTLE Technology," in *Technical Reference Manual*, 2014.
[97] A. Holey, V. Mekkat, P.-C. Yew, and A. Zhai, "Performance-Energy Considerations for Shared Cache Management in a Heterogeneous Multicore Processor," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2015.
[98] A. J. Smith, "Cache memories," *ACM Computing Survey*, 1982.
[99] P. Panda, G. Patil, and B. Raveendran, "A survey on replacement strategies in cache memory for embedded systems," in *IEEE Distributed Computing, VLSI, Electrical Circuits and Robotics (DISCOVER)*, Aug 2016.
[100] S. G. Chandar, M. Mehendale, and R. Govindarajan, "Area and Power Reduction of Embedded DSP Systems Using Instruction Compression and Re-configurable Encoding," in *IEEE/ACM International Conference on Computer Aided Design. ICCAD 2001. IEEE/ACM Digest of Technical Papers (Cat. No.01CH37281)*, 2001.
[101] C. Lefurgy, P. Bird, I. C. Chen, and T. Mudge, "Improving Code Density Using Compression Techniques," in *Proceedings of the International Symposium on Microarchitecture (MICRO)*, 1997.
[102] C. Lefurgy, E. Piccininni, and T. Mudge, "Reducing Code Size with Run-time Decompression," in *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, 2000.
[103] M. Ros and P. Sutton, "Code Compression Based on Operand-factorization for VLIW Processors," in *Data Compression Conference, 2004. Proceedings.*, 2004.
[104] N. Viennot, E. Garcia, and J. Nieh, "A Measurement Study of Google Play," *SIGMETRICS Perform. Eval. Rev.*, 2014.
[105] H. Wang, Z. Liu, Y. Guo, X. Chen, M. Zhang, G. Xu, and J. Hong, "An Explorative Study of the Mobile App Ecosystem from App Developers' Perspective," in *Proceedings of the 26th International Conference on World Wide Web*, 2017.
[106] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter, "A Study of Modern Linux API Usage and Compatibility: What to Support when You'Re Supporting," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.
[107] K. Hazelwood and A. Klauser, "A Dynamic Binary Instrumentation Engine for the ARM Architecture," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.
[108] Google, "Android Studio," https://developer.android.com/studio/index.html, 2017.
[109] X. Zhang, N. Gupta, and R. Gupta, "Whole Execution Traces and their use in Debugging," *The Compiler Design Handbook: Optimizations and Machine Code Generation*, 2007.