Fine-Grain Program Snippets Generator for Mobile Core Design

Shuang Song, Raj Desikan*, Mohamad Barakat*, Sridhar Sundaram*,
Andreas Gerstlauer and Lizy K. John
The University of Texas at Austin, Austin, TX, USA
*Samsung Austin R&D Center, Austin, TX, USA
songshuang1990@utexas.edu,{gerstl,ljohn}@ece.utexas.edu,
{r.desikan,m.barakat,s.sundaram}@samsung.com

ABSTRACT

As mobile devices have come to dominate modern computing systems, improving mobile processors' performance and energy consumption has become an important topic for both industry and academia. Similar to desktop and server CPUs, the design of mobile processors relies on a combination of high-level micro-architectural and low-level RTL simulations for design exploration and optimization. Shrinking simulation times remains a critical design issue, as it directly impacts turnaround time and time-to-market. Many prior works have been proposed to generate smaller program snippets (simulation points) aimed at reducing the simulation times for the micro-architecture design process. However, as emerging mobile applications are divergent from traditional desktop/server workloads in terms of functionality, code length, and code composition, prior strategies may no longer be satisfactory. In this paper, we propose a novel approach to generate fine-grain application snippets of mobile benchmarks that can accurately represent the full application performance for both high-level micro-architectural and low-level RTL simulations. Compared to prior work, snippets produced by our generator reduce performance estimation error by 5.6% while saving 76% simulation time on average.

1. INTRODUCTION

In the era of mobile computing, processor design still plays an essential role, as it can significantly impact device performance, energy consumption, and user satisfaction [8]. Therefore, improving mobile CPU performance and energy efficiency is considered as a primary goal [8][14][15][11] for industry and academia. Similar to conventional CPU design, design of mobile CPU design relies on high-level micro-architecture and low-level RTL simulations for design space exploration and implementation. Hence, reducing simulation time becomes extremely important to keep design time manageable. At the same time, mobile applications differ from desktop and server applications in terms of service functionality, software length, and code composition. This causes significant differences in micro-architectural behavior, such as instruction cache misses/stall cycles and branch mis-predictions [7].

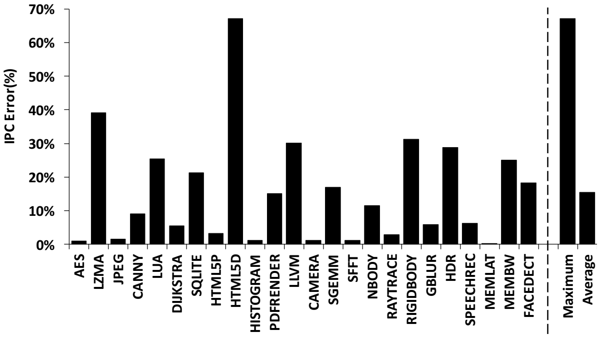


Figure 1: Errors in IPC for 24 emerging mobile applications with prior approach [13].

Many prior work [10] [13] [6] have proposed sampling-based methodologies to represent full benchmark by using sampled program snippets. Although such work can significantly reduce snippet length, they still require simulating hundreds of millions of instructions. This limits the number of micro-architecture explorations that can be performed even with fast micro-architectural simulators. Furthermore, this also makes RTL-based explorations practically infeasible. RTL simulations are slow. In practice, they have to finish over night, where only smaller segments of sampled snippets generated by prior work or completely different micro-benchmarks with affordable instruction lengths are used by industry. However, this can lead to mismatches between high-level micro-architectural and low-level RTL simulations. Such correlations are necessary to help diagnose potential implementation bugs and reveal performance/energy issues.

To make use of the existing sampling-based approach [13] for both micro-architecture and RTL simulations, the granularity of snippets needs to be reduced from the 100M to the 100K instructions level. However, as shown in Figure 1, program snippets generated by prior work [13] at a granularity of 100K instructions (with 100K-instruction warmup) cause unsatisfactory performance estimation errors for emerging mobile applications. The maximum instructions per cycle (IPC) error is around 67% and the average error reaches 15.4%. The inaccuracy of performance estimation can result in misleading design decisions.

In this paper, we present a novel approach for automatic generation of fine-grained snippets of mobile applications to speed up the simulation time of mobile CPU designs. Compared to prior works, our approach does not solely rely on code structures (basic blocks) to characterize program phases. Therefore, even when the basic block distribution becomes sparse at fine-granularity, we can

GLSVLSI '17, May 10-12, 2017, Banff, AB, Canada
© 2017 ACM. ISBN 978-1-4503-4972-7/17/05. . . $15.00
DOI: http://dx.doi.org/10.1145/3060403.3060439

still achieve better clustering. In addition, to achieve comprehensive program characterization with better accuracy, we add more micro-architecture independent features that are specific to emerging mobile applications. The specific contributions of this paper are as follows:

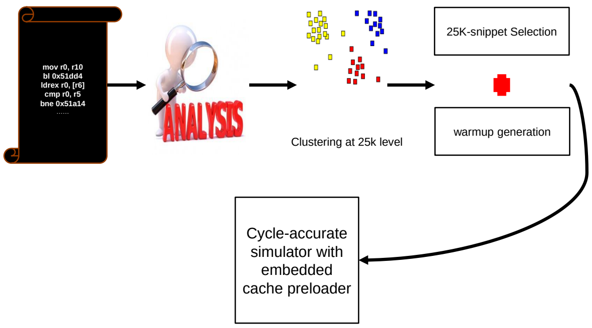
- We propose a novel generator that leverages principal component and clustering analysis to capture representative program snippets of mobile applications at fine-granularity, and use them to substitute the full trace in the simulation process. These finer-grained snippets can be applied for both high-level micro-architecture and low-level RTL simulations, which help hardware designers correlate the micro-architecture simulation to the RTL one. Compared to prior work, the snippets provided by our generator can reduce the average performance estimation error from 15.4% to 9.8% while saving 76% simulation time.

- In order to precisely characterize mobile applications, we define domain-specific features to capture unique characteristics of mobile workloads, such as percentage of CRYPTO instructions and memory misaligned accesses. Those provided features are micro-architecture independent, so they can be reused for mobile CPU designs in the future.

- Moreover, due to the length limit of generated snippets, we analyze the impacts of warmup and preload for fine-grained benchmark snippet generation on state-of-the-art mobile CPUs. This analysis can guide performance studies to generate fine-grain snippets from other applications for modern mobile CPU designs.

2. RELATED WORK

In this section, we want to review several prior works that are relevant to this paper. Simpoint [13] method makes the use of basic block distribution to find small portions of the application to simulate that can represent the full program's behavior. This approach is heavily based upon the program's code structure. Each simpoint/snippet is collected at the granularity of 100 million instructions for SPEC95 and SPEC2000 benchmarks. However, if the snippet granularity is reduced by 1000 times, the basic block distribution will become very sparse, which is really difficult to cluster. Our approach does not solely depend on the code structures (basic blocks), so it achieves better accuracy for mobile applications at fine-granularity.



```
mov r0, r10
bl 0x51dd4
ldrex r0, [r6]
cmp r0, r5
bne 0x51a14
......
```

Figure 2: The flow overview of the proposed generator.

Eeckhout et al. [6] define 47 micro-architecture independent features based on instruction mix, working set sizes, etc., to characterize the conventional CPU applications (from SPECint2000 and SPECfp2000) at the level of 100 million instructions per interval. We extend, provide, and use more mobile application specific features, as we target on these emerging applications for mobile processor designs. Different from Simpoint work, Eeckhout et al. study the similarities among all applications, and select the snippets to represent the entire benchmark suite instead of an individual benchmark. These chosen snippets are used to perform design space exploration, which requires a huge number of executions to cover various micro-architectural configurations. Since the program length and simulation time of these conventional benchmarks are much longer than the modern mobile applications', we do not necessarily have to perform this further reduction among all applications. Additionally, most mobile application suites score processor's performance based on the execution speed of each application, like [1]. Therefore, it is necessary for the hardware designers to monitor the performance of each workload, when optimization is applied on the design.

Other than these two prior works, Gutierrez et al. [7] discover the differences between SPEC CPU2006 benchmarks and smartphone applications, and reflect the poor performance of instruction cache, instruction TLB, and branch predictor. Lau et al. [10] study the applications to identify the program phases at the variable length. Since we target at 25K-instruction granularity, snippets from a giant repeated loop will all be collected. Dhodapkar and Smith [5] found a relationship between phases and instruction working sets, as phase changes occur when the working set changes. We include multiple instruction working set sizes as micro-architecture independent features in our generator. SMARTS [16] employs full functional warmup during the entire non-sampling units in-between sampling units, which provides nearly perfect warmup effects. However, it significantly limits the time reduction due to the long warmup, which occupies more than 99% of the entire simulation time. Conte [4] deploys the adaptive functional warmup. Compared to them, our generator provides a just-right warmup trace and only preloads for the necessary components. Jiang and Yu [9] target on the optimization of warmup for multi-thread simulations, such as PARSEC benchmarks [2], which is different from our emphasis.

3. METHODOLOGY

In this section, we first give an overview of proposed generator's flow, then describe the details of deployed micro-architectural independent features, snippet selection, and warmup/preload trace generation.

3.1 Generator Flow

The input of our generator is the program trace. As shown in Figure 2, we split the trace into 25K-instruction snippets, then apply predefined micro-architectural independent features to characterize these snippets. Snippets with similar features will be clustered in one group, and the snippet located in the center of the cluster will be selected to represent the cluster. Corresponding warmup and preload traces are generated for each selected snippet.

3.2 Program Features

Eeckhout et al. proposed 47 micro-architecture independent features to characterize the conventional CPU applications, such as SPEC CPU benchmarks. In our work, we make use of these characteristics and extend them with more mobile-specific features. These added features are categorized in several groups, such as instruction mix, instruction level parallelism (ILP) for load and store instructions, and misaligned accesses. All the features used in our methodology, including the previous and our new ones, are summarized in Table 1 for integrity.

Table 1: List of micro-architectural independent fingerprinting parameters used to characterize mobile workloads.

Category	Fingerprints	Explanation
Instruction mix	Load, store, control, arithmetic	% load, store, control, arithmetic instructions
	Float, SIMD, CRYPTO	% floating-point, single instruction multiple data, crypto instructions
Instruction level parallelism (ILP) for load&store	LLP_W-32, 64, 128, 256	ILP counts for load&store instruction in 32, 64, 128, 256 instructions window
Instruction level parallelism (ILP)	ILP_W-32, 64, 128, 256	ILP counts in 32, 64, 128, 256 instructions window
Register traffic	RegNOp	average number of operands
	RegTc_1	% of instructions with 1 target register
	RegTc_2	% of instructions with 1< target register
	RegUse	average degree of freedom
	RegD_1, 4, 8, 16, 32, 64	pro register dependency = 1, <= 2, <= 4, <= 8, <= 16, <= 32, <= 64
Working set	D-WS_32, 64, 128, 4096	number of data stream within 32, 64, 128, 4096B block level
	I-WS_32, 64, 128, 4096	number of instruction stream within 32, 64, 128, 4096B block level
Data stream stride	IcLdS_0, 8, 16, 32, 64, 512, 4096	pro local load stride = 0, <= 8, <= 16, <= 32, <= 64, <= 512, <= 4096
	lcStS_0, 8, 16, 32, 64, 512, 4096	pro local store stride = 0, <= 8, <= 16, <= 32, <= 64, <= 512, <= 4096
	glLdS_0, 8, 16, 32, 64, 512, 4096	pro global load stride = 0, <= 8, <= 16, <= 32, <= 64, <= 512, <= 4096
	glStS_0, 8, 16, 32, 64, 512, 4096	pro global store stride = 0, <= 8, <= 16, <= 32, <= 64, <= 512, <= 4096
Branch predictability	GAg	PPM predictor global history, global predictor
	GAs	PPM predictor global history, local predictor
	PAg	PPM predictor local history, global predictor
	PAs	PPM predictor local history, local predictor
Misaligned access	ld-ma-8, 16, 32, 64, 4096	Pro load 8B, 16B, 32B, 64B, 4096B miss aligned
	st-ma-8, 16, 32, 64, 4096	Pro store 8B, 16B, 32B, 64B, 4096B miss aligned

3.2.1 Instruction mix

We include the percentage of load, store, control, arithmetic, floating point, SIMD, and CRYPTO operations in this category to fully analyze the instruction mix of mobile workloads. Compared to the prior work, SIMD and CRYPTO are the new metrics we added. For mobile workloads leveraging vector instructions, such as SGEMM and SFFT, SIMD operation is heavily used in major program phases. CRYPTO instruction is largely deployed in AES workload, which is not widely used in traditional SPEC benchmarks.

3.2.2 ILP for load&store instructions

Load and store instructions become more important for the mobile devices, as the sizes of caches and memory subsystem are limited but highly related to the performance/user experience. Therefore, we dedicate four fingerprinting features to quantify the amount of ILP for only load & store instructions. We assume an idealized out-of-order mobile processor model, where all components are idealized and all resources are unlimited except for the instruction window. These added features measure the number of independent load & store instructions located in the instruction window, which has four different sizes (32, 64, 128, and 256).

3.2.3 Misaligned access

Misalignment happens when read or write operations try to read N bytes of data starting from an address that is not evenly divisible by N. Different architectures attempt to perform the misaligned memory accesses in different manner. For example, some processors raise exceptions when unaligned access is detected, and some are able to perform them transparently. However, none of them can avoid the performance degradation for such memory access. For mobile processors, these existing misalignments in the mobile applications can easily impact the application's execution speed. Therefore, we consider misalignment as one of important performance factors. In our application characterization, we separate load & store instructions, and categorize these accesses by the size of data they are attempting to touch. As the capacity of mobile processor's caches and memory grows, our feature set contains the misalignment of 8B, 16B, 32B, 64B, and 4096B for both load and store instructions.

3.2.4 Other features

Similar to Section 3.2.2, window size is the only tunable parameter of ILP metrics. Different from Section 3.2.2, ILP metrics quantify the amount of ILP for all types of instructions (not only load & store). All the register traffic, working set, and data stream strides are included in the prior work. Register traffic characteristics are used to capture the number of input operands, the average number of times a register is consumed, and register dependency distance. Working set features count the number of unique blocks (32B, 64B, 128B, 4096B) in each interval. Data stream analyzes both local and global data strides. The branch behavior is also an important characteristic we want to capture. Same as prior work, we want to measure how predictable the branches are for a given execution interval. To accomplish this, we deploy the Prediction by Partial Matching (PPM) predictor [3], which is built on the notion of a Markov predictor. 'G' stands for global branch history, and 'g' means one global predictor table. 'P' stands for local branch history, and 's' means an individual table for each branch. PPM is a theoretical basis for predicting branch prediction, therefore, these features still hold the characteristic of being micro-architectural independent.

3.3 Snippet Selection

To select the most representative snippets for each benchmark, we need to study the similarity of split snippets in order to avoid choosing repeated ones. We leverage two statistical methods, which are the Principal Component Analysis (PCA) and K-means Clustering. Both of them are widely used in the performance evaluation and benchmark analysis domain [12].

3.3.1 Principal Component Analysis

After characterizing all snippets using predefined program features, we have to group the snippets in multiple clusters and select the representative one from each cluster. Due to the size of the deployed program feature set, the number of dimensions for clustering becomes too high to produce a limited number of compact clusters. Therefore, we leverage the PCA method to remove the correlation among these features. Since we are not studying similarities across benchmarks, the snippets of each trace may end up having different sets of important features after the PCA process. For example, SFFT application with a high percentage of floating point (fp) instructions has the percentage of fp instructions as one of principal components, where the integer applications will not consider it as an important characteristic. For all mobile applications in this work, the generator selects the reduced set of principal components to cover more than 90% of the program variances. Other than the 90% minimal bound requirement, we add one more constraint in the generator. We define the component with more than 1% program variance coverage as an important feature to include. Therefore, some applications end up having more principal components than necessary to cover 90% variance, as we want to incorporate all the important features. On average, each application has 14 private principal components.

3.3.2 Clustering Analysis

After we identify the principal components, we can move on to the clustering phase in the generator. We use the K-means clustering algorithm, as we can directly control the number of clusters by setting the *k* value. Of course, other clustering algorithms with the same functionality can substitute the K-means algorithm. In this phase, the issue we are facing is how to choose the minimal number of clusters that can provide good clustering quality. We deploy a metric called 'cubic clustering criterion' (CCC) to quantify the clustering quality. The CCC value is proportional to the clustering quality, as higher CCC means better quality. To achieve better quality, increasing the number of clusters is the most effective way. However, as the number of clusters (*k* value) can directly determine the number of snippets to be selected for simulation. This is considered as a tradeoff problem. Therefore, we provide two knobs in the clustering phase of the proposed generator, which are the qualified CCC value and maximum *k* value (the maximum number of clusters). From our experience, most of the applications reach robust CCC value (> 800) with less than 10 clusters. Thereafter, we set the maximum number of clusters for each workload to be ten.

3.4 Warmup/Preload Trace Generation

Due to the length limitation, most generated snippets suffer from the impact of cold misses in multiple hardware components, such as multi-level caches. Prior work does not face this issue, as their snippets are long enough that negative influences due to the lack of warmup cannot affect the performance significantly. However, for fine-grained snippets, warmup can cause severe performance impact, which will be demonstrated in section 4. To avoid this problem, our generator can automatically extract the warmup trace for each snippet. The warmup trace is defined as the section of instructions located prior to the target snippet. For example, after the snippet *A* is selected from its cluster, the generator uses its beginning instruction to trace back certain number of instructions and extract those instructions to form the corresponding warmup trace. For our snippets, the length of warmup trace is set to be 100K-instruction (same as the baseline we are comparing to).

Even though the warmup trace can indeed help reflect the true performance of snippets, it has a non-negligible disadvantage, as it still requires to be executed on the simulator, which can be expensive if the length is long. However, after we diagnose the state-of-the-art mobile processor designs, we recognize that the multi-layer cache system is a critical component that needs a long warmup trace. Therefore, we deploy the preloading technique and implement the preloader on top of our in-house micro-architectural simulator to generate the trace for preloading. Similar to warmup trace capturing, we extract a section of code prior to the snippet and feed them to our preloader. The preloader will execute the trace and dump the image of multi-layer caches into a yaml file. This cache image file can be preloaded back to both micro-architectural simulator and RTL simulator before executing the warmup trace and representative snippet. Since we only need to generate this preloading file once and it can be reused, the overhead can be ignored.

4. EXPERIMENTS AND RESULTS

In this section, we would like to review the experiment infrastructure and selected mobile applications, discuss the insightful observations, and analyze the experimental results.

4.1 Experiment Infrastructure

To demonstrate the accuracy and effectiveness of our generator, we evaluate it with 24 emerging mobile applications. These 24 benchmarks are categorized into four groups, which are cryptography workloads, integer workloads, floating-point workloads, and memory workloads. The information about each application's type can be found in Table 2. Each of them is deployed as a single-core application. All of them are implemented in C++ and compiled by Clang compiler. The performance data is collected on the in-house micro-architecture simulator, which simulates the Samsung Exynos M1 mobile processor. The preloader is embedded in the simulator to snapshot the cache status image at any given moment. In addition, the simulator can take the cache image as an input and reside the status in the multi-layer caches before executing the snippet. The baseline (named as Prior_work in the figures) we are comparing to is the 100K-instruction program snippets generated from Simpoint method [13] associated with 100K-instruction warmup.

Table 2: 24 Emerging mobile workloads classification [1].

Type	Applications
Cryptography workloads	AES
Integer workloads	LZMA compression, JPEG compression, Canny, Dijkstra, SQLite, LLVM, HTML 5 Parse, HTML 5 DOM, PDF rendering, Lua, Camera
Floating point workloads	SGEMM, SFFT, Gaussian blur, Ray trace, N-Body physics, Rigid body physics, HDR, Histogram equalization, Face detection, Speech recognition
Memory workloads	Memory bandwidth, Memory latency

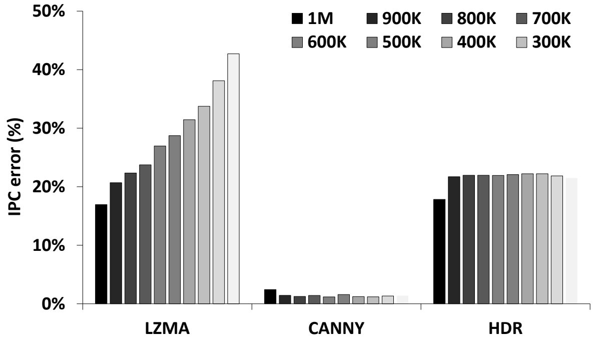


Figure 3: Warmup sensitivity study for LZMA, CANNY, and HDR.

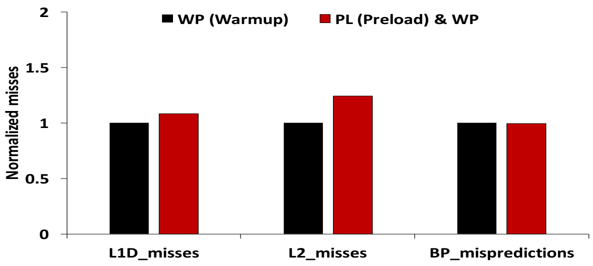


Figure 4: Component-level performance comparison between solely using warmup trace and using combination of preloading and warmup tarce.

4.2 Mobile Applications

Cryptography workloads: The Advanced Encryption Standard (AES) is the only cryptography workload, which defines a symmetric block encryption algorithm. It is widely used to secure communication channels and information. This workload encrypts a 32MB string using AES running in CTR mode with a 256-bit key.

Integer workloads: LZMA is lossless compression algorithm, while JPEG is a lossy image compression. LZMA compresses and decompresses a 450KB HTML ebook using SDK as the implementation of core algorithm. Compared to traditional bzip2 benchmark in SPEC, LZMA features a high compression ratio. JPEG works by encoding an image in 8 * 8 blocks, and each block is transformed using a discrete cosine transform (DCT). Canny is a typical sophisticated technique in image processing and computer vision, which contains four components, such as noise removal (Gaussian Blur), gradient calculation, minimum maximum threshold pass, and edge following (BFS). The Dijkstra workload computes driving direction between destinations. The data for this workload is captured from Open Street Map data for the Waterloo region, which includes 79392 nodes and 162644 edges with integer weights. SQLite is a self-contained SQL database engine that executes queries against an in-memory database. This SQLite benchmark is created to stress the underlying engine using a variety of SQL features and query keywords, such as SELECT, COUNT, SUM, etc. The LLVM workload processes an LLVM IR file through its optimizer and code-generation routines. The IR file is generated from a 3,900 line C file using Clang. HTML 5 Parse and DOM constructs a parse tree from a HTML5 document using Gumbo, and creates a HTML5 Document Object Model (DOM), respectively. Both of them stress the multi-layer cache system in the mobile CPU. PDF rendering parses and renders a 29-page PDF document, which contains mostly text with a few small images. Lua executes a Lua script using standard Lua interpreter to parse data from a JSON file, and uses Mustache to combine data to produce an HTML file. Camera replicates a photo sharing application like Instagram that merges several functions into one, such as AES, Lua parsing, JPEG, and SQLite. All these functions are executed on the CPU without GPU support.

Floating point workloads: The SGEMM (general matrix multiplication) workload computes the result of C = AB + C, where A, B, and C are matrices with 512 * 512 single precision. GEMM is implemented using vector instructions, such as AVX, SSE2, SSE3, and ARMv8 NEON. SFFT executes an FFT algorithm on an 32MB input in 16KB chunks. Similar to SGEMM, it also deploys vector instructions. Face detection and HDR are from the computer vision domain. The Face detection includes scale, pose, occlusion, expression, makeup, and illumination operations. Gaussian blur benchmark blurs an image using Gaussian spatial filter with a fixed sigma of 1.0f. This sigma translates into a filter diameter of 9 pixels. Histogram equalization workload is designed to improves the contrast in an image by implying mapping one distribution to another. It contains four steps, which are image loading, image converting, equalizing histogram, and displaying equalized image.

Memory workloads: The Memory Latency benchmark measures the latency of system memory by traversing a circular linked-list. The nodes in the linked-list are arranged to reduce the TLB misses to the level of 1 miss per page. The Memory Bandwidth captures the sustained memory bandwidth by using vector instructions, such as AVX and ARMv7 NEON.

4.3 Warmup Sensitivity Study

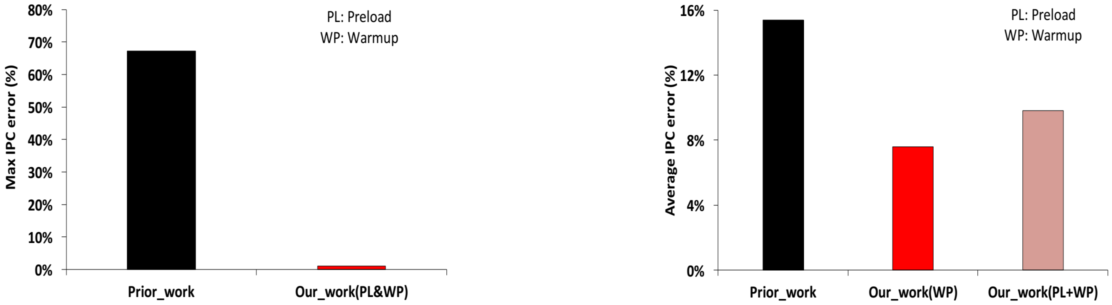
In this experiment, we attempt to quantify application's sensitivity to the length of warmup trace. Since our snippets are 25K-instruction each, we perform this study only with the warmup trace ranging from 1M-instruction to 100K-instruction. As shown in Figure 3, a different application has different reactions to the warmup length. LZMA snippets can be concluded that they are extremely sensitive to its warmup traces, as the application is doing compression work that needs to exercise the L1 and L2 caches heavily. Therefore, with 1M warmup instructions, LZMA snippets achieve the minimal IPC estimation error. However, compared to LZMA, CANNY and HDR are much less sensitive. CANNY snippets' performance is very stable with various warmup lengths, and HDR snippets lose 4% accuracy by decreasing 1M warmup instructions to 100K instructions.

4.4 Warmup vs. Preload

To reduce the long warmup trace for applications that are sensitive, such as LZMA, we deep dive into the performance analysis of LZMA. As we mentioned above, LZMA is a compression application that stresses the multi-level memory subsystems during execution. Therefore, we deploy the preloading technique to resolve this issue. Our preloader can snapshot the cache system at any given time and preload it back. As shown in Figure 4, we can see that using a 900K preload image with 100K warmup trace results in similar number of misses in the L1 data cache and L2 cache. Moreover, we observe another interesting behavior, as the 100K warmup can approximately reach the same level of performance for the branch predictor, which has some storage components like branch history tables. Similar behaviors are also captured in other applications so that we can claim using 100K warmup should be able to provide enough exercise to the major core components. Caches need to preload the status image in order to reduce the negative impacts of modeling the performance of modern mobile processor.

4.5 Performance Estimation and Simulation Time Reduction

The maximum error generated by prior work appears in the HTML 5 DOM application. The reason behind it is HTML 5 DOM needs a long time to warmup the 'cold' cache hierarchies. As we discussed above, long warmup trace is too expensive for simulations. Our implemented preloader indeed helps solve this problem. Figure 5a shows that we reduce the max error from 67.2% to 1.0%. We evaluate our generator across all 24 applications. As shown in Figure 5b, our generator with 1 million instructions warmup can significantly decrease the average performance estimation error by half (around 7.6%), however, 1 million instructions are too costly in terms of simulation time. Using a low overhead preloading technique for 900K instructions with 100K warmup can provide an average error of 9.8%. Compared to the prior work, our snippets are extremely fine-grained, so the number of instructions for the simulation has been saved by 76%, as Figure 6 illustrates. The number of instructions to be simulated is strongly proportional to the simulation time.



(a) Maximum IPC error reduction on HTML5DOM workload.

(b) The comparison of average errors in IPC between prior work and our work.

Figure 5: Comparing the IPC estimation errors of our proposed work to the errors caused by Simpoint for 24 emerging mobile workloads.

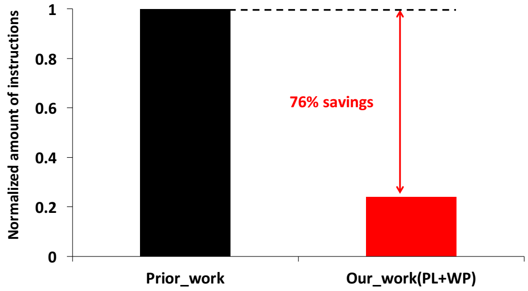


Figure 6: Comparison of the number of simulated instruction needed for Simpoint and our work.

Hence, we expect that our snippets can reduce the simulation time significantly as well.

5. CONCLUSIONS

In this paper, we propose a novel snippet generator that can produce representative program snippets for emerging mobile applications at fine-granularity. Our generated snippets are short enough to fit in both RTL and microarchitecture simulators. Compared to prior work, snippets provided by our generator reduce the average performance estimation error from 15.4% to 9.8% across 24 emerging mobile workloads. Meanwhile, these snippets shrink the total simulation time by up to 76%, which benefits the mobile processor's time to market. Furthermore, we define mobile-specific program characteristics that can be leveraged to analyze the mobile applications for other embedded research areas. Lastly, we study and quantify the mobile applications' sensitivity to the warmup length and propose the use of cache preloading technique to replace the long warmup trace.

6. ACKNOWLEDGMENTS

This work was supported in part by Samsung GRO grant, and National Science Foundation grant CCF-1337393. Any opinions, findings, conclusions, or recommendations are those of the authors and do not necessarily reflect the views of these funding agencies.

7. REFERENCES

[1] Geekbench 4 CPU Workloads. Technical report, Primate Labs, 08 2016.

[2] C. Bienia, S. Kumar, et al. The parsec benchmark suite: Characterization and architectural implications. In *PACT*, 2008.

[3] I.-C. K. Chen, J. T. Coffey, et al. Analysis of branch prediction via data compression. In *ASPLOS VII*, New York, NY, USA, 1996. ACM.

[4] T. M. Conte, M. A. Hirsch, et al. Reducing state loss for effective trace sampling of superscalar processors. In *ICCD '96*. IEEE Computer Society, 1996.

[5] A. S. Dhodapkar and J. E. Smith. Managing multi-configuration hardware via dynamic working set analysis. In *ISCA '02*. IEEE Computer Society, 2002.

[6] L. Eeckhout, J. Sampson, et al. Exploiting program microarchitecture independent characteristics and phase behavior for reduced benchmark suite simulation. In *IISWC*, 2005.

[7] A. Gutierrez, R. G. Dreslinski, et al. Full-system analysis and characterization of interactive smartphone applications. In *IISWC*, 2011.

[8] M. Halpern, Y. Zhu, et al. Mobile cpu's rise to power: Quantifying the impact of generational mobile cpu design trends on performance, energy, and user satisfaction. In *HPCA*, 2016.

[9] C. Jiang, Z. Yu, et al. Shorter on-line warmup for sampled simulation of multi-threaded applications. In *ICPP '15*. IEEE Computer Society, 2015.

[10] J. Lau, E. Perelman, et al. Motivation for variable length intervals and hierarchical phase behavior. In *ISPASS '05*, Washington, DC, USA, 2005. IEEE Computer Society.

[11] X. Li, G. Yan, et al. Smartcap: User experience-oriented power adaptation for smartphone's application processor. In *DATE '13*, San Jose, CA, USA, 2013. EDA Consortium.

[12] A. Phansalkar, A. Joshi, et al. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *ISCA '07*. ACM, 2007.

[13] T. Sherwood, E. Perelman, et al. Basic block distribution analysis to find periodic behavior and simulation points in applications. In *PACT '01*, Washington, DC, USA, 2001. IEEE Computer Society.

[14] Y. Shin, H. J. Lee, et al. 28nm high-k metal gate heterogeneous quad-core cpus for high-performance and energy-efficient mobile application processor. In *ISOCC*, 2013.

[15] S. Swanson and M. B. Taylor. Greendroid: Exploring the next evolution in smartphone application processors. *IEEE Communications Magazine*, 2011.

[16] R. E. Wunderlich, T. F. Wenisch, et al. Smarts: Accelerating microarchitecture simulation via rigorous statistical sampling. In *ISCA '03*. ACM, 2003.