Workload Synthesis: Generating Benchmark Workloads from Statistical Execution Profile

Keunsoo Kim, Changmin Lee, Jung Ho Jung, and Won Woo Ro
School of Electrical and Electronic Engineering, Yonsei University, Seoul, Korea
{keunsoo.kim, exahz, jace.jung, wro}@yonsei.ac.kr

***Abstract*—We propose an approach for benchmark workload generation. The proposed *workload synthesis* generates *synthetic workload*s that model the behavior of real applications. Statistical execution profile of a workload is constructed from hardware performance counters available in recent processors, and the overhead of profiling is significantly lower than instrumentation or simulation which requires inspection of instruction stream. Workload synthesis can be applied even though the source codes or binaries of real applications are not available, because it utilizes only statistical profile. In addition, for non-deterministic workloads, using synthetic workloads provides more consistent results than executing real workloads, since a synthetic workload replays predetermined instructions reconstructed from the execution profile of a real workload. Furthermore, with a sampling technique, we can reduce the execution time of synthetic workloads while preserving its run-time characteristics. We have implemented and evaluated the proposed method on ARM-based mobile devices. The results show that synthetic workloads reproduce the profiled performance event counts of real workloads with high accuracy.**

I. Introduction

Benchmarks are indispensable tools to investigate various aspects of computer systems. System designers employ various benchmark applications and observe their runtime characteristics to empirically optimize performance, execution time, power dissipation, and energy consumption. Real applications are often used as benchmark workloads. In this case, the true behavior of the target system is observed to acquire realistic results. A typical example is the SPEC benchmarks [1], which consists of real workloads representing the typical applications of workstations and servers.

However, there are practical limitations in benchmarking with real applications. Because they have become interactive and consume considerable amount of time outside of the application code, it becomes more difficult to develop portable benchmark methods that reflects the behavior of real applications correctly. This is because recent applications require specific components or functions of a system, and need to organize the input in a specific manner [2]. For instance, let us consider a case of designing benchmark workload from an interactive 3D mobile game in which most of the inputs are provided by a touchscreen or sensors. In the real situation of running the game, touch screen and GPU acceleration are required. In fact, when the workload is simulated, the simulator should model all components and functionalities to correctly execute subject workloads, which significantly increase simulation complexity and burden of modeling. In addition, the run-time behavior of interactive or real-time applications are affected by input, which is sensitive to timing and context. When evaluating on a real system, the subject application may interact with UI component or hardware accelerators which increase non-determinism. Therefore, this type of applications may not show consistent execution behavior over different runs. In this case, applying a pre-recorded user input scenario for a specific execution (in this example a sequence of touch events) is not sufficient to acquire consistent results. A possible solution is to modify the application to remove the nondeterminism. Unfortunately, such modification is not always possible; one reason is that simply the source code is not available, particularly in case of commercial applications.

However, if we are only interested in the behavior of specific components, modeling all system components may not be necessarily required. In this sense, we can imagine a hypothetical device that records and segregates the behavior of specific aspects of the running workload from its execution, and replays them deterministically. This is conceptually similar to record and replay (RnR), which has been studied for debugging or auditing purposes [3]–[6]. Regarding the aforementioned difficulties, we believe that RnR is also useful in generating benchmark workloads from real applications.

Unfortunately, implementing such a RnR device with low cost and low overhead is a challenging work, because of limited observability on the subject application. For example, we can instrument the subject application process to extract its instruction stream while the application is run on a real system [4], [7]–[10]. However, instrumentation is an intrusive method that significantly increases the execution time of the subject. This disrupts time-critical actions, such as responding for touch-screen events, therefore it is not suitable for interactive or timing-sensitive workloads. An alternative is to simulate the workload [10]. However, in addition to difficulty in modeling all hardware devices for functional correctness, it is significantly slower than executing on a real machine and user interaction with simulated applications is difficult to be performed. Another solution that minimizes the overhead is to attach special hardware modules on the real machine running the subject application in order to capture machine states without affecting the execution [6]. However, it also requires dedicated hardware modules and equipments.

In this paper, we propose an alternative approach to overcome the limitations of the existing methods, and demonstrate that it is achievable on a real machine. In our approach, the execution of a real application is traced through the hardware performance counters of a processor, which are designed to profile microarchitectural events such as number of executed instructions, cache and branch misses, *etc.* Such statistical information on execution characteristics of the subject (*i.e.*, statistical execution profile) can be collected with low overhead and it does not severely affect the execution of the subject workload.

978-1-4799-6454-3/14/$31.00 ©2014 IEEE

We have designed a modeling technique to generate an executable program that replays the recorded from the collected execution profile, which we name *workload synthesis*. The statistical characteristic of the generated workload (*synthetic workload*) behaves similarly to the original one in terms of performance event counts in the statistical profile. In this approach, we can selectively generate the activities of specific system components from the execution profile of an application. In addition, synthetic workloads are deterministic in terms of the executed instructions, since it executes only predetermined instruction streams. Furthermore, we have developed a technique that significantly reduces the benchmark execution delay with synthetic workloads by employing partial execution techniques similar to the sampling simulation methods [11]. By leveraging the similarity between real and synthetic workloads and the acceleration technique, we target to use synthetic workloads as the surrogates of real applications, and will develop a portable benchmark suite that estimates the performance of various real applications rapidly.

We have implemented the proposed technique targeting high-performance mobile devices such as smartphones or tablet PCs. The experimental results demonstrated that the proposed workload synthesis is able to reproduce performance event counts with high accuracy (average error of 2.8%). In terms of IPC, the average error is 21%, however we observed both relatively small error for some workloads (error of -6%) and an outstandingly different case (error of 41%).

The contributions of this paper are summarized as follows:

- We present the concept and implementation of workload synthesis that replays the profiled execution characteristics (hardware performance counters) of real workloads.
- We make a case of the workload synthesis targeting ARM-based mobile smartphones. We demonstrate the proposed method can be implemented non-invasively on a real system with low overhead utilizing hardware performance counters.
- We evaluate workload synthesis using real-world applications with real smartphones. The results show that synthetic workloads reproduce performance counter event profile of real workloads with high accuracy, and various IPC error values are observed.
- We also show that execution speed of synthetic workloads can be accelerated while preserving the characteristics.

II. Workload Synthesis

A. Overview

Figure 1 illustrates the workflow of the workload synthesis. The workflow is composed of two stages. In the sampling stage, the time-varying execution characteristic of the target application is recorded. The target application is monitored by the profiler. The profiler samples the number of event occurrences (event counts) on the runtime activities of the target application from various information sources, including hardware performance counters and the OS kernel. The event counts are sampled periodically to keep track of time-varying characteristics of the application. Generally the event counts are accumulative, therefore we need a preprocessing step to extract the number of event occurrences during one sampling interval.

For convenience, we name a collection of event counts occurred during a sampling interval as a *workload slice*. We define the sampling interval as *slice length*, and it decides the time domain resolution in profiling. The workload slices are arranged in the execution time order to form the *tracing log* which contains the complete time-varying execution characteristics of the application. Figure 2 illustrates how a tracing log is constructed from workload slices and samples of time-varying event counts.

In the regeneration stage, a synthetic workload is generated from the tracing log by the workload synthesizer module. The synthesizer inspects each workload slice in a tracing log. It assembles synthetic workload from the code templates called *kernel function*, which is a sort of microbenchmark codes designed to perform the task corresponding to an event field in a workload slice. Finally, the generated synthetic workloads can be executed as benchmark workloads, and the characteristics of real and synthetic workloads can be compared to validate the synthetic workload.

B. Performance Counter Events on Mobile Platform

We selected microarchitectural performance events available in the performance monitoring unit (PMU) of the ARMv7 processors, as listed in Table I. The events are selected regarding the availability in the mobile devices we have used. The number of executed instructions (*NInst*) is further divided into three different types: ALU (*ALU*), branch (*BR*), and memory operations (*MEM*). ALU indicates the simple ALU operations such as MOV, ADD, and SUB. MEM is the number of load and store instructions, which is estimated from the number of L1D cache accesses. BR indicates the number of executed branches. The idle time (*Idle*) of the workload is also computed from the task clock (*TC*), which is the net execution time of the target thread, to reflect the wait time behavior. The number of cache misses (*CM*) is estimated by L1D cache refill count. The number of branch misses (*BM*) is also tracked. We used *perf* and *procfs* interfaces provided by the Linux kernel to access the hardware performance counter in user mode.

C. Assumptions and Limitations

Cache and branch misses are dependent to microarchitecture implementations. Therefore, the number of misses may change under the different hardware configurations because they depends on the data access pattern of a program. However, the information on the pattern is not available from statistical event counts, and therefore this is an inherent limitation of statistical profiling. Nevertheless, the miss counts must be included in the workload synthesis process, because the execution characteristics of synthetic workloads may be distorted significantly without considering them. To avoid such difficulties, we assume no extreme change in average number of cache or branch misses even though different systems having different microarchitectures are evaluated. Then, in the workload synthesis process we treat the misses as if they are the characteristics of a workload and generate the same number of misses in the profile.

In profiling performance statistics, currently only one thread of the application can be monitored, due to the limited number of simultaneously manageable hardware performance

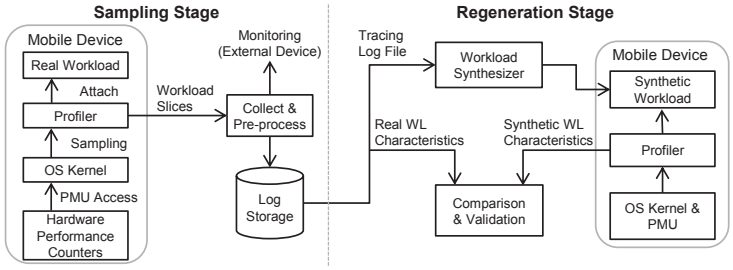


Fig. 1: Workflow of workload synthesis on mobile devices

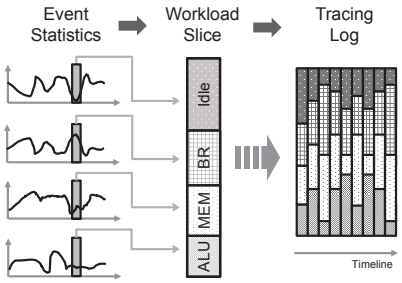


Fig. 2: Tracing log generation

TABLE I: Summary of profiled fields

Field	Source
Total Instrs (*NInst*)	PMU
ALU Instrs (*ALU*)	*NInst - BR - MEM*
Memory Instrs (*MEM*)	PMU (Estimated from L1 cache access count)
Branch Instrs (*BR*)	PMU
Idle Time (*Idle*)	Sampling Interval - *TC*
Cache Misses (*CM*)	PMU (Estimated from cache refill count)
Branch Misses (*BM*)	PMU
Task Clock (*TC*)	OS Kernel

counters of ARMv7 processor we used [12]. In fact, *perf* provides an emulation mechanism to share performance counters among multiple threads, with time-division scheduling of the counters. However, we found that the accuracy of the counted values is reduced significantly when the counters are shared, therefore we avoided to use it. The recent Intel processors support per-core monitoring and provide more performance counter events, and we may be able to utilize such features to profile multithreaded applications. We left further development of the proposed scheme regarding other platforms as a future work.

III. Generation of Synthetic Workloads

A. Overview

A tracing log is just a collection of event counts for a certain execution of a workload. Therefore, in order to acquire an executable program as workload from the event counts, we utilize kernel functions as building blocks for a synthetic workload. For a given tracing log, the mappings between kernel functions and event counts are computed for every workload slices to compose the synthetic workload.

Our observation is that workloads are only distinguishable through the number of occurrences for all events. In other words, if there are multiple workloads which result in the same event counts with the given counts, then they are indistinguishable. Based on this principle, the kernel functions are assembled so that the execution of kernel functions generates the same number of event counts recorded in workload slices. We control the event counts of kernel functions by allocating iteration numbers to each kernel function.

Figure 3 illustrates a concrete example of how the kernel functions are associated to event counts in a tracing log, which is performed by the workload synthesizer module. First, the characteristic of each kernel function, which is defined as the number of events occurred with one iteration of the kernel function, should be known to the solver. For each workload slice, the synthesis solver computes the required iteration

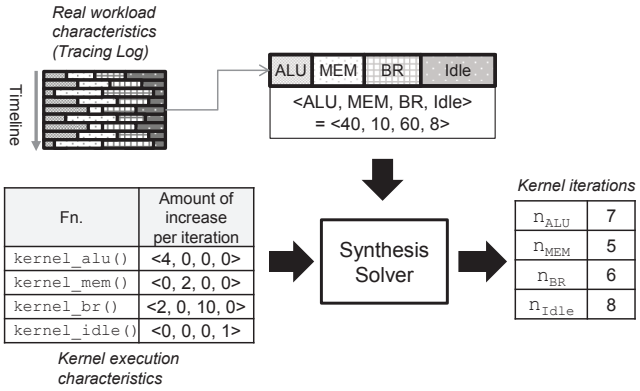


Fig. 3: Operation of the workload synthesizer

number of each kernel function to match the event counts of the current workload slice.

B. Workload Synthesis Algorithm

In this section, we discuss the details of the synthesis algorithm. We first formalize the concepts that we have discussed. Let $c_{i,j}$ indicate jth counter value of the ith workload slice in a tracing log. When there are N_j data fields, for the ith workload slice, the characteristic vector $\mathbf{r_i}$ is defined as an ordered set of the counter values as the following.

$$\mathbf{r_i} = [c_{i,0} \quad c_{i,1} \quad ... \quad c_{i,N_j}]^T \tag{1}$$

For instance, for the six counter fields listed in Table I, the subscript j can be mapped to each counter field as follows:

$$\mathbf{r_i} = [c_{i,ALU} \quad c_{i,BR} \quad c_{i,Mem} \quad c_{i,Idle} \quad c_{i,BM} \quad c_{i,CM}]^T \tag{2}$$

Note that for an execution of a kernel function with iteration number 1, the characteristic vector of the execution can be identified, by treating the kernel function as a special type of workload. To clarify, we denote the characteristic vector of a kernel function k as $\mathbf{f_k}$, as shown in the following equation.

$$\mathbf{f_k} = [c_{k,0} \quad c_{k,1} \quad ... \quad c_{k,N_j}]^T \tag{3}$$

Kernel characteristic matrix F is defined by collecting characteristic vectors for all N_k kernels as the following.

$$F = [\mathbf{f_0} \quad \mathbf{f_1} \quad ... \quad \mathbf{f_{N_k}}] \tag{4}$$

A concrete example of the characteristic matrix F in case of six kernel functions and six counter fields is shown below.

$$\begin{bmatrix} c_{ALU,0} & c_{ALU,1} & c_{ALU,2} & c_{ALU,3} & c_{ALU,4} & c_{ALU,5} \\ c_{BR,0} & c_{BR,1} & c_{BR,2} & c_{BR,3} & c_{BR,4} & c_{BR,5} \\ c_{Mem,0} & c_{Mem,1} & c_{Mem,2} & c_{Mem,3} & c_{Mem,4} & c_{Mem,5} \\ c_{Idle,0} & c_{Idle,1} & c_{Idle,2} & c_{Idle,3} & c_{Idle,4} & c_{Idle,5} \\ c_{BM,0} & c_{BM,1} & c_{BM,2} & c_{BM,3} & c_{BM,4} & c_{BM,5} \\ c_{CM,0} & c_{CM,1} & c_{CM,2} & c_{CM,3} & c_{CM,4} & c_{CM,5} \end{bmatrix} \quad (5)$$

Note that F contains complete information of kernel functions, and is irrelevant to workload slices.

For ith workload slice, we define the kernel iteration vector $\mathbf{p_i}$ that contains the iteration numbers of each kernel function $n_{i,k}$ as the following.

$$\mathbf{p_i} = [n_{i,0} \quad n_{i,1} \quad ... \quad n_{i,N_k}]^T \quad (6)$$

Let $\mathbf{r_i}'$ be the characteristic vector after executing for all kernel functions with $n_{i,k}$ times. Then, for ith workload slice, the workload synthesis algorithm should compute $\mathbf{p_i}$ from $\mathbf{r_i}$ such that $\mathbf{r_i}'$ is equal to $\mathbf{r_i}$. In other words, the synthesis algorithm should find out all $n_{i,k}$ satisfying the following relation:

$$\mathbf{r_i}' = \sum_{k}^{N_k} \mathbf{f_k} \cdot n_{i,k} = \mathbf{r_i} \quad (7)$$

We can rewrite the above equation in terms of the kernel characteristic matrix F and $\mathbf{p_i}$ as the following equation.

$$F \times \mathbf{p_i} = \mathbf{r_i} \quad (8)$$

The synthesis solver should compute $\mathbf{p_i}$ of the above equation to determine the appropriate iteration numbers for each kernel. However, computing $\mathbf{p_i}$ is not trivial. First, F can be non-invertible matrix because of the non-orthogonality of the kernel characteristic vectors. Second, F can be a non-square matrix if the number of kernel functions exceeds the number of the profiled fields. Therefore, we obtain pseudo-inverse of F to compute least square approximation of $\mathbf{p_i}$ from $\mathbf{r_i}$ as follows.

$$F^T F \times \mathbf{p_i} = F^T \mathbf{r_i} \quad (9)$$

$$\mathbf{p_i} = (F^T F)^{-1} F^T \mathbf{r_i} \quad (10)$$

Finally, the solver computes (10) for all workload slices, and the computed $\mathbf{p_i}$ vectors are stored as the result of workload synthesis.

C. Orthogonality of Kernel Functions and Synthesis Coverage

For a given combination of event counts in a workload slice, the synthesis solver computes the combination of kernel iteration numbers with (10). However, it can be computed if and only if F has full column rank. Regarding the definition of F, it implies that we need the kernel functions at least as many as the number of events in a workload slice. For instance, the number of kernel functions should be at least six to cover the counter fields listed in Table I.

In addition, (10) does not give the exact solution if F is non-invertible. It gives the exact solution when $\mathbf{f_k}$ are orthogonal basis functions. In other words, it is preferred that each kernel function increases the count of only one event field. In this analogy, we define such property as *orthogonality* of kernel functions.

```
void kernel_alu(int N);    /* ALU ops */
void kernel_br(int N);     /* Branch hit */
void kernel_mem(int N);    /* Load/Store (L1 hit) */
void kernel_idle(int N);   /* Idle time (1 us) */
void kernel_bmiss(int N)   /* Branch miss */
void kernel_cmiss(int N); /* L1 miss */
```

Fig. 4: List of kernel functions

With non-orthogonal kernels, it is likely that we get non-exact $\mathbf{p_i}$ solutions with large errors which cause significant misbehavior of the synthetic workloads. To keep the error in a tolerable range, we adopt a filtering mechanism. For every workload slice, the synthesis error e is computed as the following.

$$e_i = ||\mathbf{r_i}' - \mathbf{r_i}||^2 \quad (11)$$

Then the filter will ignore the results that $e_i > \lambda$, where λ is a predetermined error threshold. Therefore, some slices may not be included in the synthetic workload, which means the synthetic workload covers the less number of workload slices compared to the real workload. In this sense, we define synthesis coverage as a ratio of the workload slices which are not ignored by the filter.

We observed that as the 'angle' between kernel characteristic vectors decreases (i.e., as the vectors become similar) the coverage is dropped significantly, because of the frequently occurred large errors.

IV. Implementation of Kernel Functions

As discussed, orthogonality of kernel functions is important to maintain the synthesis coverage high. However, implementing orthogonal kernel functions is not a trivial work. Assume we design an orthogonal kernel function for MEM; it should not execute any branch or comparison operations because it is supposed to run only memory access instructions. This means the function cannot check iteration count and therefore, it may be repeated infinitely. As such, some non-orthogonality is unavoidable. However, we should minimize its effect as possible, by carefully designing the operation of kernel functions.

A. ALU, BR, MEM, and Idle Kernel Functions

Figure 4 lists the prototypes of the kernel functions. In common, the number of iterations is passed as the argument. To repeat the operation with the specified iteration number, each kernel function contains several compare and branch instructions, and this comparison reduces orthogonality. To minimize its effects, we unroll the function body as possible. For instance, with unroll factor of 10, each kernel executes ten kernel operations and one comparison operation. Therefore, the side-effects of the comparison task is reduced to the order of magnitude. However, the unroll factor should not be large, because it reduces the granularity of workload generation; in the previous example, the event counts can be reproduced only with the unit of 10s. We empirically determined an appropriate unroll factor for each kernel function.

`kernel_alu()` repeats the general integer arithmetic operations which consist of the mixture of MOV, ADD, and SUB operations. This kernel function is designed to match the

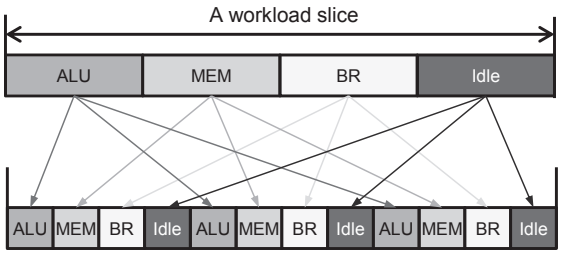


Fig. 5: Workload slice mixing

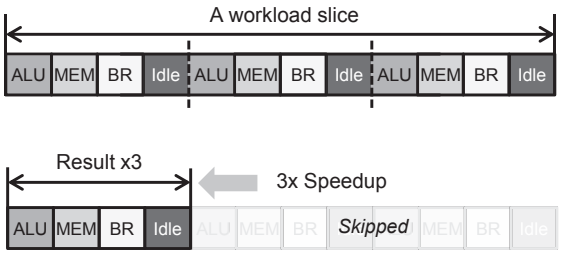


Fig. 6: Benchmarking time acceleration

overall ALU operation counts because all ALU instructions are profiled collectively in the mobile system we used. If they are individually counted, we can further decompose the kernel into `kernel_mov()`, `kernel_add()`, *etc.* This may better model the real workload.

`kernel_br()` and `kernel_mem()` performs branch and memory operations, with branch prediction and cache access are hit, respectively. As similar to `kernel_alu()`, `kernel_mem()` performs uniformly mixed load and store operations. `kernel_idle()` make the current thread sleep for the number of microseconds.

B. Branch Miss and Cache Miss Kernel Functions

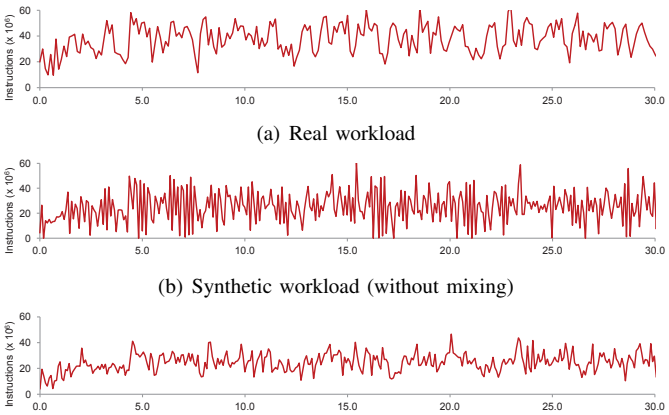
`kernel_bmiss()` generates branch misses. In the kernel function, the branch instructions form a goto chain so that the branch target of each instruction is another branch instruction. The target address of each branch is specified by a ring-shaped jump table which contains the randomly shuffled addresses of the next branch instruction. In the jump table, there is a special entry that branches to the subroutine which checks exit condition and rotates the jump table ring. By rotating the ring, each branch instruction read different branch targets for every iteration. Therefore, a branch miss is occurred in every branch execution because the branch target cannot be correctly predicted.

The design of `kernel_cmiss()` is inspired by LM-Bench [13]. In our implementation, the kernel function generates cache misses with pointer chasing among randomly assigned memory locations. As the preparation step, an array of which the size is sufficiently larger than the last level cache size is allocated. Each entry of the array stores pointers (i.e., indices of the array), which are chased by reading the next accessing index from the array. To generate cache miss, the order of indices are shuffled to be accessed randomly. We select the next index so that the address distance between indices larger than the cache line size. This shuffling is performed before the pointer chasing is initiated.

V. Execution Techniques for Synthetic Workloads

A. Slice Mixing

Figure 5 presents the concept of the slice mixing technique. For a workload slice, each kernel functions are executed sequentially with the computed iteration numbers. In the mixing technique, we divide the iteration numbers into smaller sub-iterations, as if they are derived from the sub-slices generated with more frequent sampling (i.e., smaller slice length). The execution of the sub-slices can be interleaved as illustrated in



(a) Real workload

(b) Synthetic workload (without mixing)

(c) Synthetic workload (with mixing)

Fig. 7: The effects of workload slice mixing

the figure. We found that such redistribution of kernel iterations increases similarity of the observed execution characteristic between synthetic and real workloads. Figure 7(a) presents the executed number of instructions of the real *Sunspider* application. Figure 7(b) is the results without workload mixing, of which the result pattern looks much noisier than the real one. However, as presented in Figure 7(c), when the mixing is applied the pattern becomes smoother and more similar to the real workload.

We analyze how the mixing technique works from the perspective of signal processing. When workload slices are sampled from the execution of a real workload, the effective signal bandwidth of sampled slices becomes half of the sampling frequency, as stated by the sampling theorem [14]. Therefore, if we sample the execution of the synthetic workload generated from the real workload with the same sampling frequency, this further reduces the effective bandwidth by the half of the original one. This can be interpreted as an aliasing effect; the noisy pattern in Figure 7(b) can be interpreted as the artifacts incurred by the aliasing. Therefore, workload mixing is analogous to an anti-aliasing technique with signal up-sampling. It further divides the collected workload slices to smaller workload slices and the effect of this operation is equivalent to increasing the sampling frequency since the number of workload slices increases. In conclusion, the artifacts are removed and synthetic workloads can exhibit the more similar pattern to the real workloads.

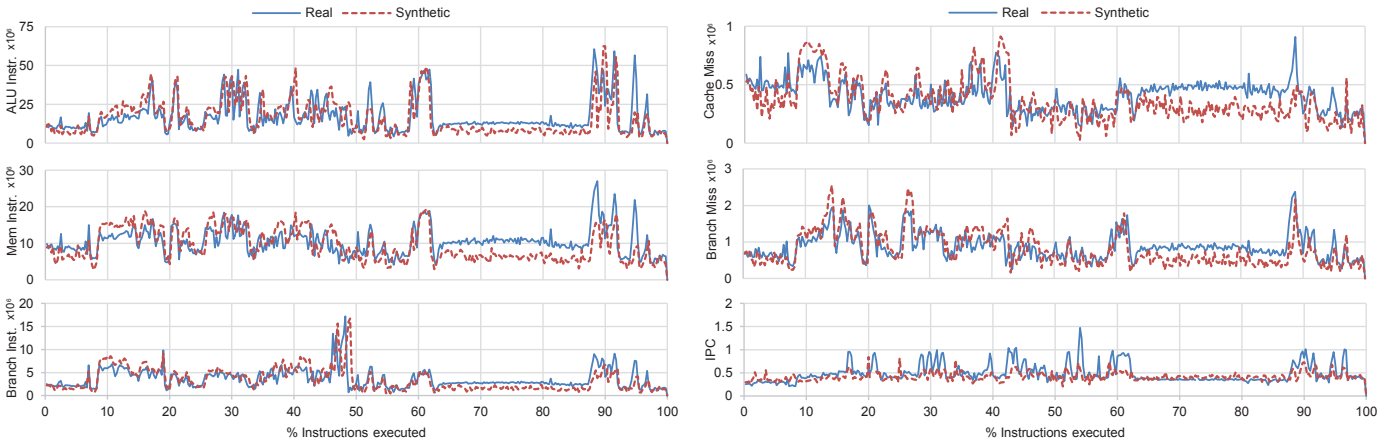


Fig. 8: Similarity of execution characteristic between real and synthetic workload (*RealRacing* on GS3)

Fig. 9: Cross-correlation coefficient between real and synthetic workload

	ALU	MEM	BR	CM	BM	IPC
GoogleMap (NexS)	0.66	0.68	0.74	0.58	0.52	0.16
GoogleMap (GS3)	0.26	0.28	0.26	0.23	0.36	0.059
RealRacing (GS3)	0.68	0.61	0.61	0.54	0.69	0.51
Sunspider (NexS)	0.20	0.16	0.15	0.087	0.09	0.19
Sunspider (GS3)	0.54	0.50	0.44	0.32	0.51	0.13
Avg.	0.47	0.45	0.44	0.35	0.43	0.21

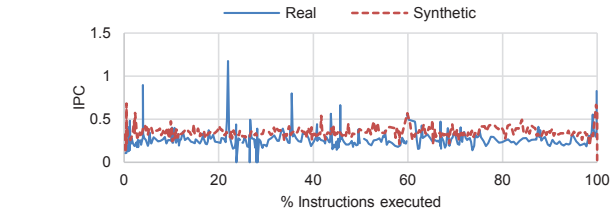


Fig. 10: A case of IPC graph mismatch between real and synthetic workload (*GoogleMap* on GS3, $r = 0.059$)

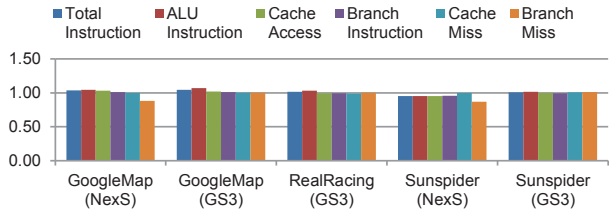


Fig. 11: Total event counts of synthetic workloads (Normalized to real workload)

B. Accelerated Execution

Workload mixing splits a large workload slice into small sub-slices. Therefore, it produces many identical workload slices of which the execution pattern is regularly repeated during the slice length of the original slice. By exploiting this regular property of sub-slices, the benchmarking time with synthetic workloads can be reduced. Figure 6 shows how the acceleration is performed. By the effect of workload mixing, the workload sub-slices are generated. As the figure shows, the sub-slices in a workload slice are equivalent. Therefore, by skipping the execution of the repeated sub-slices, we can accelerate the execution time of the synthetic workload. The full execution result can be estimated from the result of skipped execution with simple linear interpolation. For instance, in the example shown in Figure 6, the full execution characteristics are estimated by multiplying the skipping factor of 3 to the results of the accelerated workload execution.

VI. Experimental Results

A. System Setup

To evaluate the feasibility of the proposed method, we built a testbed which consists of real smartphones and applications. We used two smartphones for evaluation; Galaxy S3 (**GS3**) and Nexus S (**NexS**). Galaxy S3 has a Exynos 4412 processor containing 1.4 GHz quad-core ARM Cortex-A9, 1 GB of main memory, and Android 4.1.2 as the OS. Nexus S has a Exynos 3110 processor containing 1 GHz single-core ARM Cortex-A8, 512 MB of main memory, and Android 4.1.1 as the OS.

We used three mobile applications. **Sunspider** [15] is a computing-intensive javascript benchmark suite, which executes various JavaScript workloads on top of the default web browser of the mobile platform. As discussed, due to the hardware restrictions the profiler can profile only one thread at a time. In SunSpider, the profiler is attached to WebViewCoreThread, which is the most computing intensive thread that executes the JavaScript engine. **GoogleMap** [16] is an interactive map application. The user input is simulated with an input script that repeats horizontal swiping with the interval of one second. We profiled the main thread of the map application, which showed the highest CPU utilization. **RealRacing**3 [17] is a 3D racing game with high quality graphics, controlled by the tilt sensor and touchscreen. Because of the system requirements for the game, Nexus S was not able to run the game. We profiled the rendering thread which consistently maintained the highest CPU utilization regardless of the game state (menu or playing).

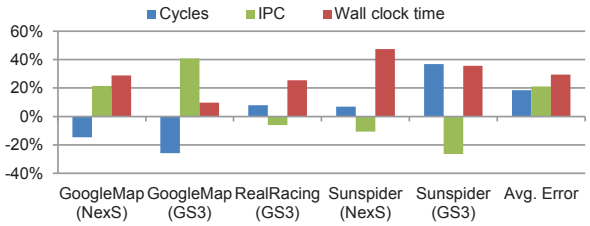


Fig. 12: Error in cycles, IPC, and wall clock time

B. Comparison of Real and Synthetic Workload

Figure 8 compares performance counter profiles of the real and synthetic workload for the *RealRacing* workload running on the GS3 device. The figure shows event counts for six fields during the execution of the synthetic and real workload. In order to avoid the timing issues which will be discussed in the next subsection, we plot the event counts with respect to the number of executed instructions. We can visually compare the number of occurred events between real and synthetic workload. This clearly shows that the workload synthesizer is able to allocate the kernel iterations from the given workload slices and kernel function characteristics, with high accuracy.

To quantify the similarity, we have computed the Pearson correlation coefficient between real and synthetic workload profiles. Figure 9 presents the coefficient values for each application-device pairs. For the first five events (ALU, MEM, BR, CM, and BM), the coefficients are in the range of 0.35 and 0.47 for the average values, which indicates that they are positively correlated. In the case of IPC, the relatively lower average coefficient value of 0.21 suggests that the proposed method is less effective in regenerating IPC. As a case, Figure 11 shows the IPC graph of real and synthetic workloads of *GoogleMap* on GS3, which shows the lowest correlation and largest IPC errors (41% as presented in Figure 12). The synthetic workload has consistently high IPC than the real workload, and this causes a large error in average IPC.

Figure 11 summarizes the total number of event occurrences of the synthetic workloads; the values are normalized to the results of the corresponding real workloads. The events listed in Figure 11 are profiled and regenerated in the workload synthesis workflow, therefore it shows the effectiveness of the workload synthesizer. On average, the error is small; the worst case is -13.1% (*Sunspider* on NexS) and the average of all cases is 2.8%.

C. Cycles, IPC, and Wall Clock Time

Figure 12 presents the error in cycles, IPC, and wall clock time in executing the synthetic workloads compared to the real workloads. These metrics indicate how much synthetic workloads are close to the real workloads. In performance perspective, we believe that cycle count is the better metric than wall clock time since it is frequency agnostic. The errors in cycle counts are from 7% to 37% of which the average of 18%, however as the figure shows, the errors in wall clock time is larger than cycle count (29% on average).

We found that the reason of the larger error in wall clock time is that DVFS behaves differently with synthetic workloads. In addition, it should be noticed that the error increases as CPU utilization of the workloads decreases. Figure 13 shows the correlation between clock frequency and

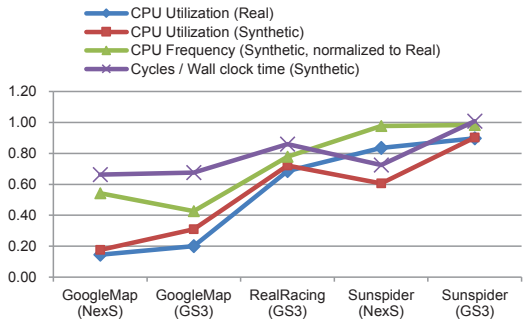


Fig. 13: CPU Utilization and clock frequency of synthetic workloads

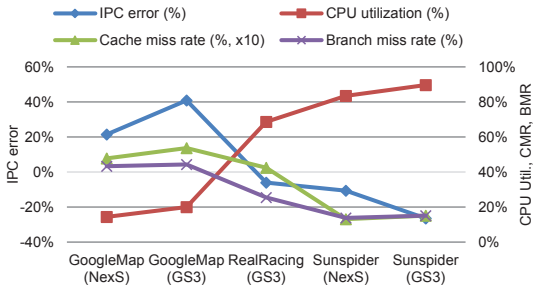


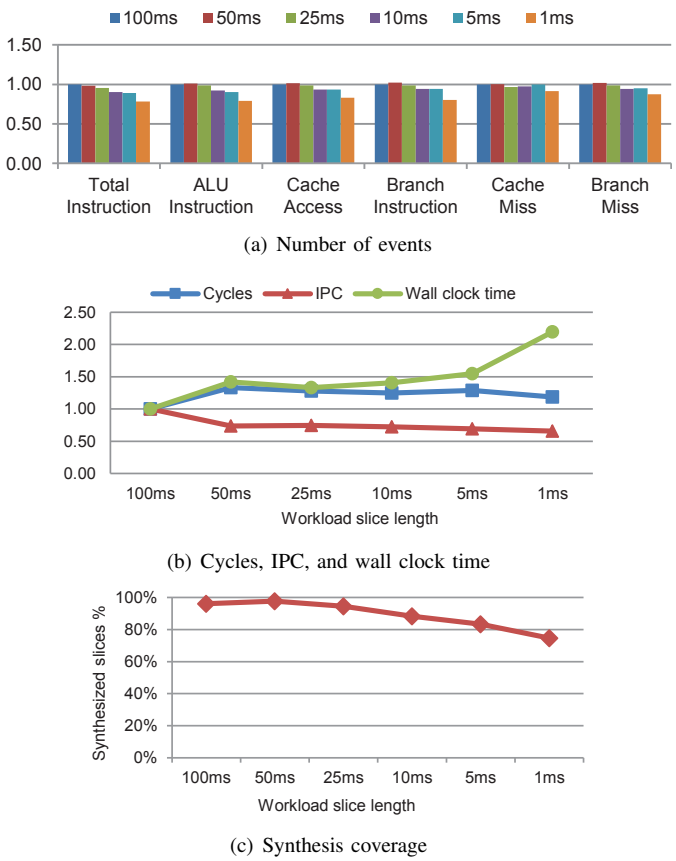
Fig. 14: Candidates of the sources in IPC error of synthetic workloads

CPU utilization, and the number of cycles executed per unit wall clock time. The figure shows that the clock frequency of synthetic workloads approaches to that of real workloads as CPU utilization increases. We suspect this is because the DVFS controller of Linux dynamically adjusts the clock frequency based on CPU utilization of all processes [18]. Therefore the frequency is sensitive to CPU utilization of the workload, however it seems that when the CPU utilization is low the DVFS performs more conservatively with synthetic workloads.

As summarized in Figure 11, synthetic workloads execute nearly equal number of instructions compared to the real workloads. Therefore, IPC is mostly determined by the cycles consumed to complete the execution. The error in IPC is from -6% to 41%, the average of which is 21%. It is interesting that the IPC errors are various and do not show any trend among all synthetic workloads. Figure 14 shows the candidate reasons that may cause the IPC error. It suggests that as cache miss or branch miss rate is increased, error in IPC with synthetic workloads is increased. In general, they significantly affects IPC and we are assuming that all cache or branch misses are uniform in synthetic workload generation, it may increase the difference in IPC as cache or branch misses dominate the cycles. If so, including additional information on cache or branch miss (*e.g.*, L2/LLC miss rate) in the statistical profile will help to reduce the IPC error of synthetic workloads. We leave the further investigation on the IPC error as a future work.

D. Dependency on Workload Slice Length

The profiler samples real applications at a unit of workload slice, of which the length is the interval that the counter values are accumulatively collected. Therefore, the length of workload slices may affect the accuracy in profiling and generation of



(a) Number of events

(b) Cycles, IPC, and wall clock time

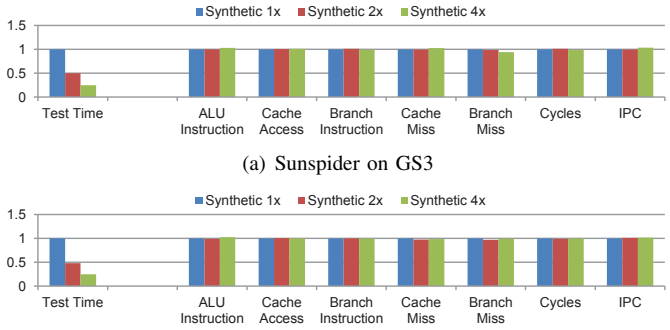
(c) Synthesis coverage

Fig. 15: Sensitivity to workload slice length (Sunspider on GS3)

synthetic workloads. Figure 15 presents the profiling results of the synthetic *Sunspider* workload on GS3 with six different workload slice lengths. In the results, the event counts tend to decrease as the length decreases. We found that the reason is because the synthesis coverage decreases as the slice length decreases. Figure 15(c) shows the coverage is decreased by 20% in the 1 ms case. This means that 20% of the slices are not included in the synthetic workload. Interestingly, Figure 15(b) shows that the wall clock time is significantly increased with 1 ms workload slice, while the cycle count and IPC is not changed. We suspect that the reason is the overhead of the idle kernel function (currently implemented with *usleep()*) for short wait time of the short slices. We believe this can be alleviated by providing the better implementation for the idle kernel function.

E. Accelerated Benchmarking

As discussed in Section V-B, benchmarking time of synthetic workloads can be reduced significantly by adopting better scheduling techniques of workload slices. Figure 16 shows the effect of the acceleration for the case of *Sunspider* on both devices, with setting the full execution of the synthetic workloads (1x) as the baseline. We can find that the benchmarking time, which is equal to the wall execution time of a synthetic workload, is reduced proportionally as the acceleration ratio is increased from 1 to 2 and 4. Furthermore, the results of all metrics, which are computed by multiplying the measured values to the acceleration ratio, are kept consistent. This means applying the acceleration technique does not affect in estimating the execution characteristics. Therefore, the results demonstrate the effectiveness of the acceleration technique with synthetic workloads, and we believe that this will save a significant amount of time when especially many applications are evaluated.



(a) Sunspider on GS3

(b) Sunspider on NexS

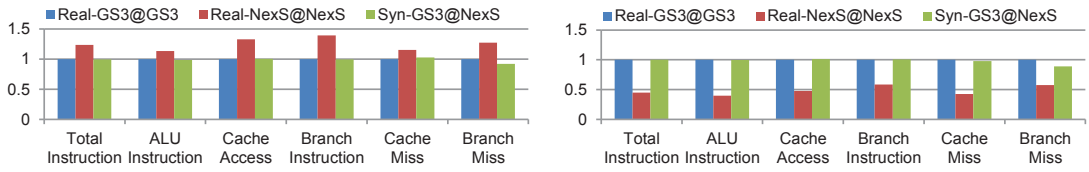
Fig. 16: Effect of benchmarking acceleration technique

F. Cross-device Analysis

We evaluate the synthetic workloads in a cross-device environment, where the real workloads were profiled on GS3 and the workloads were synthesized and executed on NexS. Figure 17 shows the event counts where the values are normalized to the real workload results on GS3. Mostly, the event metrics are different in two devices; the real *Sunspider* and *GoogleMap* workloads executed 20% more and 50% less instructions compared to GS3. This shows that keeping consistency in benchmarking over different platforms is difficult; even though we use the same application, the execution characteristic is changed. For instance, in the case of *Sunspider*, the processor executes the instruction stream of the JavaScript interpreter which implementation varies over different OS versions. However, as the figure shows, the application-dependent characteristics of synthetic workloads are preserved even in the cross-platform environment. Therefore, we can minimize the effect of the software platform for fair evaluation across platforms.

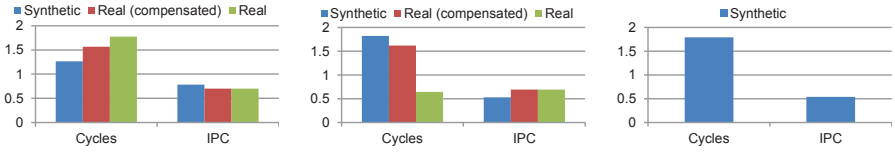
To properly function as a benchmarking workload, the synthetic workload should have distinct performance characteristics across devices, and the characteristics should be consistent to the real workload. Figure 18 compares the relative performance between NexS and GS3 with timing-sensitive metrics when real and synthetic workloads are used. All values are normalized to the real workload results on GS3. Note that the evaluated real workloads have different instruction counts on each device, which affect the results of timing-sensitive metrics. Therefore, to fairly compare the results, we present the normalized real workload results according to the instruction counts (denoted as *compensated*).

In the cases of *Sunspider* and *GoogleMap*, of which the real workload results are available, the synthetic and compensated real workloads exhibit similar trends in task execution time, cycles, and IPC. Due to the difference in instruction count, the uncompensated real and the synthetic workloads behave differently, however the synthetic workloads better follows the compensated results. Therefore, it is justified that the



(a) Sunspider (b) GoogleMap

Fig. 17: Cross-device comparison of event counts (*GS3@NexS*: recorded on GS3, replayed on NexS)



(a) Sunspider (b) GoogleMap (c) RealRacing

Fig. 18: Relative performance of NexS over GS3 (Normalized to real workloads on GS3)

synthetic workloads can be adopted for effective benchmarking across platforms. Based on the results, we can estimate the performance of *RealRacing* as presented in Figure 18(c), even if the application is not executable on the NexS device.

VII. Related Works

Benchmarks play a key role in developing and evaluating computer systems. To investigate the various aspects of the systems, a variety of benchmark suites has been developed. LMbench [13] and LINPACK [19] are code-portable microbenchmark tools to measure specific performance metrics such as FLOPS, memory bandwidth, and context switching. Our implementation of the kernel functions is inspired from them. SPEC CPU2006 [1] is a representative benchmark suite that execute the various CPU-intensive real workloads in workstations or servers, while MiBench [20] focuses on embedded system applications. Recently, many benchmark suites have been developed targeting smart device workloads [2], [21], [22]. Unlike the conventional CPU-intensive workloads, these new workloads more depend on the underlying software layer (OS or runtime systems) or hardware (GPUs or modems) [2], and the workloads often require user interactions, which makes hard to design a portable benchmark suite. A solution is to extract program traces with binary instrumentation [4], [7] and perform trace-driven evaluation. However, due to the runtime overhead of instrumentation, the execution time of the target program can increase significantly. Alternatively, we adopt the hardware performance counters to capture the time-varying application characteristic with the target application unaffected.

Through the full system simulation, the target system can be fully characterized including the effects of system software. gem5 [10] is a widely used simulator that can run full software stack with cycle level precision. However, the critical drawback of a simulation-based evaluation is long simulation time. Sampled simulation reduces the simulation time by performing the detailed simulation only for the outstanding parts of the target application. It shares the basic idea with the proposed acceleration technique. In our method, we use real applications running on real devices with real user interactions in native speed.

Statistical simulation [23]–[27] is conceptually similar to the proposed method; a synthetic surrogate of a real workload is generated with profiling, and the synthetic one generated from statistical profile is evaluated as the delegate of the real one. In statistical simulation, in general, it requires simulation of the subject application to capture the synthetic trace. For instance, in the extracted instruction trace is tagged to contain statistical metadata [23] or to specify the operation of the simulator [25], such as a cache or branch miss; which requires a special simulator that understands the tagged instructions of the synthetic traces. Meanwhile, our method is targeted on real systems; to achieve this, we leverage hardware performance counters of real machines to avoid the complexity of simulation. In addition, a synthetic workload is generated as a valid executable that can run on either a real system or simulator. Therefore, the proposed method can be adopted to unmodified real devices and can be used to capture and replay the statistical profile of applications.

Record and replay has been studied to exploit its deterministic property. For instance, replay-based debugging tools such as Flashback [3] and PinPlay [28] records the execution of the target application to debug by repeating the behavior after a crash or malfunction. This is particularly useful for parallel applications having nondeterministic bugs [28]. Paranoid Android [5] uses the replay for security auditing. Activities of a smart device are recorded and monitored to the cloud server, and replayed to check any security violations. QuickRec [6] is a hardware-assisted record and replay system, which is designed to find possible concurrency bugs in a parallel program. Although hardware-based RnR minimizes performance overhead, it requires hardware recording modules, which is the reason that we avoided it. Compared to the existing RnR methods, our method is less precise. However, we use hardware performance counters which overhead of monitoring is significantly lower than software-based inspection methods of the instruction stream.

VIII. Conclusions

In this paper, we presented *workload synthesis*, a modeling technique that reconstructs the behavior of the execution profile of real applications. The execution profiles of real applications

are collected from hardware performance counters. In our method, source codes or binaries of applications are not required to compose benchmark workloads since it depends on the non-invasively collected statistical profile of the real application. Furthermore, the complex user interactions and dependency of the underlying software framework is included in the profile, which provides a way for more consistent benchmarking of nondeterministic applications. In addition, we found that the acceleration technique is effective in reducing the workload execution time retaining the accuracy.

We designed and implemented the proposed method targeting for ARM-based smart devices. The experimental results show the synthetic workloads can replay the performance event counts of the original real workloads. We observe wide range of IPC error from -6% to 41%. In future work, we will investigate the source of the IPC difference and figure out dominating factors for the difference. The proposed method provides a systematic way to scale the number of profiled events. We have considered only few microarchitectural events due to the limited monitoring ability of the evaluated mobile system. We believe that including additional events (impact of TLB, L1/L2/LLC misses, *etc.*) possibly reduces the IPC gap. Furthermore, we will explore the applicability of the proposed method to more various workloads and systems; in particular, workload synthesis may be also useful in evaluating server workloads which incorporate complex interactions with remote systems and have long execution time.

ACKNOWLEDGMENT

This work was supported by the Application Engineering Team, Memory Business, Samsung Electronics.

REFERENCES

[1] J. L. Henning, "Spec cpu2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[2] A. Gutierrez, R. Dreslinski, T. Wenisch, T. Mudge, A. Saidi, C. Emmons, and N. Paver, "Full-System Analysis and Characterization of Interactive Smartphone Applications," in *the proceedings of the 2011 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, USA, 2011, pp. 81–90.

[3] S. M. Srinivasan, S. Kandula, C. R. Andrews, and Y. Zhou, "Flashback: A lightweight extension for rollback and deterministic replay for software debugging," in *USENIX Annual Technical Conference, General Track*. Boston, MA, USA, 2004, pp. 29–44.

[4] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, pp. 89–100.

[5] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: versatile protection for smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 347–356.

[6] G. Pokam, K. Danne, C. Pereira, R. Kassa, T. Kranich, S. Hu, J. Gottschlich, N. Honarmand, N. Dautenhahn, S. T. King, and J. Torrellas, "Quickrec: Prototyping an intel architecture extension for record and replay of multithreaded programs," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: ACM, 2013, pp. 643–654.

[7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: ACM, 2005, pp. 190–200.

[8] G. Xu, A. Rountev, Y. Tang, and F. Qin, "Efficient checkpointing of java software using context-sensitive capture and replay," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 85–94.

[9] J. Chow, T. Garfinkel, and P. M. Chen, "Decoupling dynamic program analysis from execution in virtual environments," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, 2008, pp. 1–14.

[10] 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*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[11] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS X. New York, NY, USA: ACM, 2002, pp. 45–57.

[12] *ARM Architecture Reference Manual*, ARMv7-A and ARMv7-M ed., ARM Limited.

[13] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996, pp. 279–294.

[14] A. V. Oppenheim, R. W. Schafer, J. R. Buck *et al.*, *Discrete-time signal processing*. Prentice-hall Englewood Cliffs, 1989, vol. 2.

[15] Sunspider benchmark. [Online]. Available: http://www.webkit.org/

[16] Google maps. [Online]. Available: https://play.google.com/store/apps/details?id=com.google.android.apps.maps

[17] Real racing 3. [Online]. Available: http://www.ea.com/real-racing-3-and

[18] V. Pallipadi and A. Starikovskiy, "The ondemand governor," in *Proceedings of the Linux Symposium*, vol. 2. sn, 2006, pp. 215–230.

[19] J. Dongarra and P. Luszczek, "Linpack benchmark," *Encyclopedia of Parallel Computing*, pp. 1033–1036, 2011.

[20] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. IEEE, 2001, pp. 3–14.

[21] Antutu benchmark. [Online]. Available: http://www.antutulabs.com/

[22] C. Kim, J.-h. Jung, T.-K. Ko, S. W. Lim, S. Kim, K. Lee, and W. Lee, "Mobilebench: A thorough performance evaluation framework for mobile systems," in *The First International Workshop on Parallelism in Mobile Platforms (PRISM-1), in conjunction with HPCA-19*, 2013.

[23] M. Oskin, F. T. Chong, and M. Farrens, "Hls: Combining statistical and symbolic simulation to guide microprocessor designs," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*, ser. ISCA '00. New York, NY, USA: ACM, 2000, pp. 71–82.

[24] S. Nussbaum and J. E. Smith, "Modeling superscalar processors via statistical simulation," in *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*. IEEE, 2001, pp. 15–24.

[25] L. Eeckhout, K. De Bosschere, S. Nussbaum, and J. E. Smith, "Statistical simulation: Adding efficiency to the computer designer's toolbox," *Ieee Micro*, vol. 23, no. 5, pp. 26–38, 2003.

[26] D. Genbrugge and L. Eeckhout, "Chip multiprocessor design space exploration through statistical simulation," *Computers, IEEE Transactions on*, vol. 58, no. 12, pp. 1668–1681, Dec 2009.

[27] D. Genbrugge, S. Eyerman, and L. Eeckhout, "Interval simulation: Raising the level of abstraction in architectural simulation," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010, pp. 1–12.

[28] H. Patil, C. Pereira, M. Stallcup, G. Lueck, and J. Cownie, "Pinplay: a framework for deterministic replay and reproducible analysis of parallel programs," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*. ACM, 2010, pp. 2–11.