2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)

Phase-based Profiling and Performance Prediction with Timing Approximate Simulators

Chih-Wei Yeh[1], Chia-Heng Tu[2], Yi-Chuan Liang[1] and Shih-Hao Hung[1]
[1]National Taiwan University, Taipei 10617, Taiwan
[2]National Cheng Kung University, Tainan 70101, Taiwan
Email: medicinehy@gmail.com; chiaheng@mail.ncku.edu.tw; D04922003@csie.ntu.edu.tw; hungsh@csie.ntu.edu.tw

***Abstract*—Designing a system usually acquires lots of instincts and knowledge to harmonize the computing resources under the paradigm of application specific heterogeneous systems. This paper presents a phase-based profiling mechanism to speed up the process of learning how application behaviors perform on the hardware and vice versa. By analyzing program phases, performance information can be gathered in a way that highlights the performance of high-level tasks in an application running on different hardware settings. We evaluated our phase-based profiling framework using QEMU, employing approximate timing models and mechanisms to track functions/events in programs and operating systems of the guest system. Furthermore, by using timing simulations, it is possible to escape the confined boundaries of real world machine based systems, and to rapidly explore the impact of hardware parameters on the system performance. In our experimental results, phase-based profiling yields useful information of the runtime behaviors and performance of a program, allowing developers to discover program bottlenecks, and predicts the performance of optimization ideas on the software and/or underlying hardware. Our results suggest that incorporating phase profiling with the timing approximate simulator helps to facilitate hardware and software co-design.**

I. Introduction

With the emergence of deep learning, edge computing, smart *Internet of Things* (IoT) applications and Industry 4.0, today's processing system demands more computing power than before. Heterogeneous processors and application-specific accelerators are widely used [1], [2] to achieve such a performance while maintaining the energy constraint in the *Post Moore's Law* era. Furthermore, as the complexity of such system has been increasing, the focus of designing systems has been shifted to exploring the design space with a mixture of heterogeneous processor cores/accelerators and the interconnections between the components to optimize the performance and efficiency at the system level. For each targeted application domain, the designer needs to combine different types of processor cores and makes design tradeoffs on caches, memories, and the interconnections between heterogeneous cores and accelerators.

In addition, the difficulty of fully utilizing the power of underlying hardware resources raises other challenges on choosing/porting the algorithms across accelerators. To be more specific, porting a CPU algorithm to heterogeneous platforms often needs extra care and modifications, or it may suffer from underutilizing the computing power provided by the accelerators. Also, past studies showed that application performance can be affected by the characteristics of underlying architecture, e.g. GPU branch divergence [3], GPU Memory Coalescing [4], and FPGA pipeline stall [5].

On the other hand, optimizing the applications according to the use scenario is also an important perspective, especially when the best performance/throughput/latency is required under limited computing resources. To do so, the developer needs to design the system and to choose the right algorithms/data structures. For example, database applications have column major or row major data placement which have different performance on *selection* and *update* operations [6]. In addition, in scenarios like *Facebook messenger*, a read-optimized database must have different implementations and trade-offs on the data placement, caching, logging, and etc. [7]. However, it is hard and time-consuming to understand the algorithm behaviors with existing tools.

Besides the challenges of knowing the characteristics of software and hardware, the trend of designing application specific heterogeneous systems also becomes more complex than ever. Recently, Google gives a demonstration on how machine learning can help exploring the placements of heterogeneous computing resources of deep RNN model [8]. Furthermore, Jeff Dean also proposed to revisit the heuristic in the algorithm designs in many fields with new deep learning methods [9]. These studies not only give examples on how deep learning changes the design of heterogeneous systems but also raise the needs of fast timing simulation schemes (for design space exploration) and vertical integration of how workloads affect algorithms and how algorithms perform on different hardware parameters (for understanding whether the heuristic apply to our needs). In such scenario, there is a severe need for a tool that can vertically integrate the profiling of program behaviors with the ability to explore the impacts of hardware changes on these behaviors. Furthermore, predicting and projecting the performance of running the sequential codes on different accelerators is also a key to find the best design in such a SW/HW co-design scenario.

Exploring different design parameters as well as software algorithms and data placement, architects rely on their instincts to perform cost-benefit analysis among alternative design options. However, especially in a heterogeneous computing environment, a large system has millions or billions of possibilities, and so optimizing the performance of heterogeneous applications is a challenging topic even for an experienced

2325-1301/18/$31.00 ©2018 IEEE
DOI 10.1109/RTCSA.2018.00021

programmer. To help architects explore the correlation of HW parameters and SW algorithms, timing emulators, such as gem5 [10], are often used as a timing emulator in various fields. Though equipped with many micro-architectural timing models, the existing timing emulators are lack of the analysis of software behaviors. For example, it is impossible to understand the changes of program behaviors under different workloads and the differences of performance counters when the behaviors changes. When designing a system specifically for some applications, such information is very helpful to judiciously decide the memory hierarchy, cache size and even the type of processing units (CPU/vector/GPU/FPGA).

To achieve the best performance, a designer must understand the characteristics of hardware/accelerator as well as the software since algorithms are often not designed to be portable across heterogeneous processing units due to some limitations of hardware, e.g. memory bandwidth. Though existing profiling tools [11], [12], [13] and instrumentation tools [14], [11], [15] are very powerful, such as *Vtune*, *codeXL* and *nvprof*, it is difficult or impossible to alter the hardware parameters on real machine. As for emulation tools [16], [10], [17], they are mostly too slow for modern applications such as deep learning. Although there are studies about fast and scalable emulation tools [18], [19] but they all fall in short of architectural supports such as ARM and MIPS ISA. None of the existing tools can provide both fast timing evaluation and program behavior analysis with its performance impact on various hardware parameters. Therefore, we designed a new emulation tool for such scenario based on timing approximate mechanisms so that timing model can be easily built and calibrated to new heterogeneous platforms.

In this paper, we propose a new emulation based profiling tool for heterogeneous systems, called *Snippit*. Based on the functional emulator, QEMU [20], we applied our previous studies on approximated timing emulation of hardware components [21] and emulating heterogeneous environments [22] with Multi2Sim [23]. Furthermore, by implementing the phase detection algorithm [24], *Snippit* is capable of exposing the program behaviors [25], [26], [27] with its underlying hardware counters. Lastly, we also present a machine learning model to predict the performance of each program phase running on different platforms and to provide hints of optimizations for developers, such as running on GPU/CPU.

Helping a developer finding program bottlenecks more rapidly, the proposed framework can (1) explore different hardware parameters, (2) evaluate the impact on a program phase if hardware changes, (3) discover the causes of bottlenecks and the possibility of using accelerators, and (4) understand the program's behavior. Moreover, with the integrated timing simulators, *Snippit* can provide hardware performance counters and timing information on the executed program without changing program behaviors which static/dynamic instrumentation profiling tools do. In our experimental results, we predicted the GPU-friendliness of program phases, explored the hardware designs and their relationship to resource bottlenecks, and captured the performance impacts from hardware parameters.

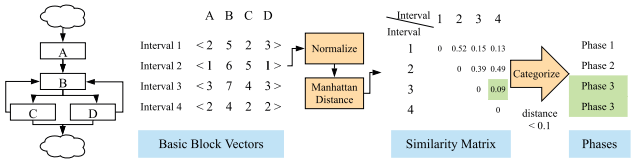

Fig. 1: The workflow of phase detection on a hypothetical program segment.

The remainder of this paper is organized as follows. Section II introduces the background of basic block vectors which is used in our phase detection algorithm. Section III describes the implementation details of the proposed phase based profiling mechanism. Section IV presents the experimental setup and results which showcase the advantages and usages of phase based profiling. Section V briefly reviews related works. Finally, Section VI presents the overall summary and findings of this paper.

II. Basic Block Vectors for Phase Characterization

Phase classification techniques typically use some form of vectors to record the execution frequencies of corresponding code segments [28], [24], [29], [30]. Basic Block Vectors (BBV) [24] that tracks the execution frequencies of *basic blocks*[1] is one of the representative examples. Usually, a BBV is composed of the frequencies of executed basic block IDs within a time interval, e.g., 100K instructions or 1K basic blocks [31], [27]. The BBVs in a phase have nearly identical values. In other words, time intervals are categorized into the same phase if they have similar BBVs[2].

Fig.1 uses a hypothetical example of control flow graph to illustrate the concept of the BBV-based phase detection method. The execution flow of a program can be divided into several *intervals* (also called *windows*) which form BBVs. For example, the frequencies of basic blocks $<A, B, C, D>$ are $<2, 5, 2, 3>$ in Interval 1. After the normalization of each BBV, the phase detection algorithm will then use Manhattan distances [32] between BBVs to build a Similarity Matrix. Finally, the algorithm will categorize the intervals (windows) into phases by a predefined distance threshold or by using a *kmeans* algorithm [33].

III. Snippit Framework Overview

As shown in Fig. 2, *Snippits* leverages the timing approximate simulation technique for modeling heterogeneous systems [34] to achieve fast simulation and provide performance counters for online phase detection. There are three main components in our proposed design. (1) The *VPMU* coordinates all HW components simulators and provides the timing information with the calculated guest time to the

[1]A basic block is a code section executed from the start to finish with only one entry and one exit.

[2]A measure of distance between BBVs is algorithm dependent.

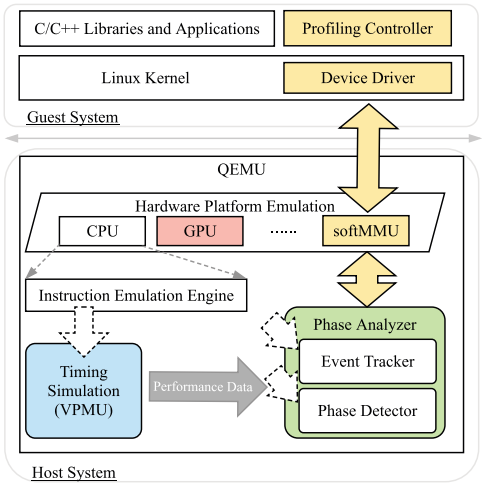

Fig. 2: Overview of the proposed *Snippit* where designs are categorized and grouped with colors.

emulated guest environment. (2) The *GPU* emulation provides both functional and timing emulation of GPU device with a generic *Accelerator Plugin Interface* which can be extended to hook any accelerators. Due to page length limitation, the details of the plugable GPU interface is not recapped here; please refer to [22] for more information. (3) The *Phase Analyzer* provides the ability to monitor and track all the function calls and events in both kernel and user space with all phase information associated to each running process. With all three components, *Snippits* is able to profile the program behavior with its detailed performance counter information and help both SW/HW designers find the bottlenecks of their applications with the hints generated from our tool.

In the following subsections, sec:timingsim briefly recaps the internals of timing approximate simulation that are useful for phase-based profiling. In sec:helpers, we describe the *Snippit* modules used for interactions between application designers/system developers and Phase Analyzer. Event Tracker and Phase Detector modules of Phase Analyzer are introduced in sec:eventtracker and III-D respectively, which are followed by the procedures for post processing of collected data in sec:postprocessing.

A. Timing Simulation

Based on QEMU [20], we adopted the timing models of our previous work, VPA [21], which provides a cycle approximate timing simulation framework with a speed of around 30 MIPS and with an 8% average error rate of MiBench [35]. In order to mitigate the impact of host simulation speed on guest behavior, we connect the virtual clock in QEMU to the timing simulator. Since guest OS relies on the clock reported from hardware (QEMU) to do all the tasks, controlling the virtual clock makes the overhead of timing simulation invisible to the guest environment.

The emulation process of QEMU consists of two phases, one is translation and the other is execution. The non-intrusive timing simulation of hardware components is done by instrumenting helper functions when performing the binary translation phase in QEMU. Instead of basic blocks (BBs), QEMU uses translation blocks (TBs) as the unit of binary translation for maintaining the state of the CPU and co-processors. In general, translation blocks are similar to basic blocks but with more limitations which would make a TB usually shorter than a BB. During the execution of each TB in the Instruction Emulation Engine, helper functions will also be executed and will send the traces to the External Architecture Models for timing simulations. In combination with the timing timulations, *Snippit* is able to obtain the performance data and the states of CPU/GPU via VPMU at any time for the phase detection and event tracking, as illustrated at the bottom of Fig. 2.

B. Interactions between Guest and Host Environments

Several modules have been developed to pass the commands from user-space in the emulated guest system to the Snippit framework in the host system. The introduction to the modules is listed as below:

- *Profiling Controller.* As illustrated in Fig. 2, users can perform performance profiling of an application with the *profiling controller*, which opens the VPMU device and passes the data to VPMU. Furthermore, Snippit users can specify the program to be monitored and what to profile; in particular, we implemented the *profiling controller* equipped with various options similar to Linux *perf*. Besides, the controller is able to control the timing simulations, such as the configuration of the guest HW platform to be simulated.
- *Device Driver.* As the software interface exposed to the Profiling Controller, the device driver provides the means of accessing the emulated VPMU device in the host environment from the guest environment. In addition, the driver is responsible for handling the symbol resolve problems across different Linux kernel version. For example, during the initialization process, it will obtain the addresses of Linux kernel symbols via *kallsyms_lookup* to cope with the problem, where the addresses to the kernel symbols would be different across kernel versions. This is important for the kernel space profiling.
- *softMMU.* We utilized the *softMMU* and *page tables* to facilitate the efficient data transfer between the Profiling Controller and the emulated VPMU. With proper address translation, Profiling Controller can transfer data to the host environment (i.e., emulated hardware platform) by just passing a pointer, instead of performing redundant data copy through kernel memory. After obtaining the pointer, *Snippit* will read the data out by accessing the pointer with *softMMU* and the address to the translation table stored in CPU state.

C. Event Tracker

In order to collect branch events of target programs and form BBVs on the host side, we developed an event tracking

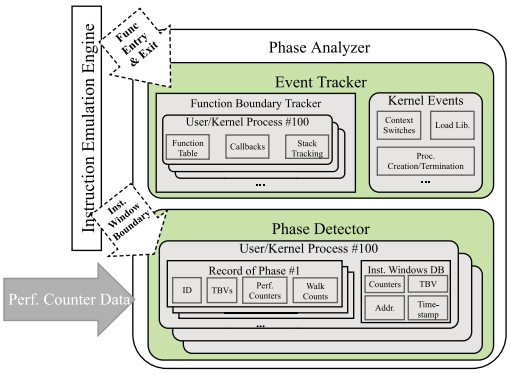

Fig. 3: Internals of Phase Analyzer.

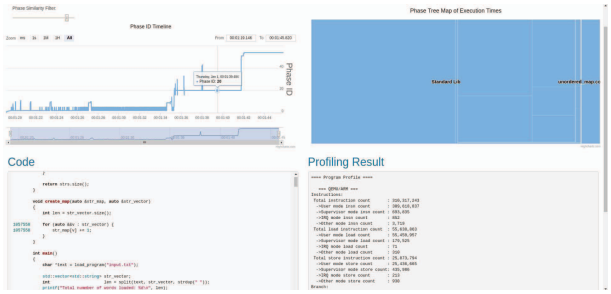

Fig. 4: The profiling tool for timeline analysis, a treemap of code proportions, a walk count with line mapping, and performance counters including estimated execution time.

mechanism inspired by SET [36] which can track Linux kernel events and function calls of processes. The implementation of *event tracker* enables collecting performance counters and phase information of the profiled program only when the target application is running.

We implemented *function boundary tracking*[3] to track function calls/returns with the ability to retrieve values of input arguments without instrumentations. To perform tracking functions and events, during the translation stage of QEMU, *event tracker* checks the start address of every TB and compare it to the corresponding function table of the current process. If the address is found in the table, this TB will be marked as a start of that function and invoke pre-defined/registered callbacks, e.g. monitoring process context switches. During the execution, marked TBs will then perform callbacks if a callback function is assigned.

In addition, we implemented all the callbacks for handling important Linux kernel events. In order to dynamically load the symbol addresses of a Linux kernel, *Snippit* gets symboal addresses from our *Device Driver* or parses the symbol table from *vmlinux*, the binary of a Linux kernel without compression. In this work, the following callbacks are implemented to accomplish the tasks of tracking processes independently which conquer one of the biggest disadvantages of using full-system emulation tool, per-user-processes information.

- *Context Switch.* In Linux kernel, *__switch_to*, is the core function to perform the context switch. By monitoring the function call and reading the input argument, we can monitor the PID of the next running process.
- *Loading binary and libraries.* To find the actual addresses where binaries and libraries are placed, we implemented a callback to both the entry and exit event of *mmap_region* for parsing input arguments and retrieving return value if the mapped memory is *executable*. With the current PID number and the file path from input arguments, we successfully track all the libraries with their start addresses.
- *Launching a process.* In launching a new process, the PID will first be stored when *wake_up_new_task* is called, then the name of the executed program will be retrieved from input arguments when *do_execv* is called. Finally, the *event tracker* will record the file path and address of the loaded binary and push the process to the tracing-list when *do_mmap* is called.
- *Terminating a process.* When *do_exit* is called, the target process will be removed from the tracing list and save all the data correlated to this process to disks.

D. Phase Detector

Since BBVs provide good sensitivity and low performance variation [38], we decided to use phase classifiers with BBVs instead of other phase detection techniques. Referencing the design in Scarphase [27] for its efficiency, we used the same technique to build our phase detection mechanism. In the implementation of Scarphase, a phase is a cluster of windows containing similar BBVs where the limit of a window is counted by a fixed number of instructions. To fit the code translation scheme on QEMU, we modified the design of BBVs to be TBVs (Translation Block Vectors).

As depicted in Fig. 3, during the execution of the target program, Phase Detector will automatically collect the start address of translation blocks and form a TBV in the current window. Also, the start and end address of each TB will also be stored in the current window to be used for line mapping source codes during post-processing. When the number of instructions exceeds the fixed limit of the window size, say 10K instructions, Phase Detector will collect the performance counters stored in the current window and then classify the phase of the current window. The classification is done by calculating the Manhattan distance between the TBV and each phase and comparing the distance to a threshold, which we set to 1 in order to minimize the variance in each phase. If the TBV does not match to any phase, a new phase will be generated. After matching the phase, the start-end address stored in the window will also be merged to the phase and will record the phase number in the timeline with a timestamp. After the termination of a monitored process, start-end addresses stored in each phase will be parsed into a file-

[3]This work reimplemented the codes of our previous work, the details of design philosophy is in our previous publication [37].

and-line map with walk counts. The information of program counter to line number is provided by parsing the DWARF segment in the profiled binary. After the parsing, Event Tracker will save window timelines, performance counters, and phase results to files for post-processing.

E. Post Processing for Phase Profiling

Merging Similar Phases. In our case study, we found that multiple phases could share the same code sequence when the window size is small. In some cases, a phase might contain too many lines of codes doing similar tasks, when the code size is large, e.g. Linux kernel. In addition, in order to analyze the trend of program behavior and identify execution stages easily, we use the combination of file name and line number of each phase as the unique key to merge similar phases. In this paper, the similarity between two phases is defined as the union divided by the summation of the size of two sets. Then, we merge phases which have a similarity greater than the user defined threshold. Merging similar phases by its lines of codes can produce a clean result with the intention of developers.

Visualization Tool. A visualization tool is important when exploring the performance issues in a complex scenario presenting a massive amount of information, as occur in phases. One of the key features of our proposed tool is that it incorporates program phase analysis with profiling information. Shown in Fig. 4, the visualization tool provides a timeline view of phase (top-left), a treemap of the files in this phase (top-right), source codes with walk count labeled next to the codes (bottom-left), and the profiling information associated with this phase (bottom-right). A user can select any phase on the timeline to explore the differences between phases or drill down the treemap to explore the proportion of codes in different files. In a highly refactored code, e.g., Linux, the treemap could help a lot on understanding what this phase was executing about.

IV. Experimenal Results for Phase Profiling

Several experiments have been facilitated to demonstrate the potentials of both software and hardware optimizations with the implemented framework in this section. The proposed framework models the Linux/ARM system, where the system configurations are given in Section IV-A. In addition, we demonstrated the effectiveness of utilizing and analyzing phase behaviors and elaborated the functionality of the phase based profiling in Section IV-B. To show that our framework is capable of analyzing the program behaviors, we use the *word count* implementing with different dictionary type and discuss the phase profiles captured by *Snippit* in Section IV-C. In addition, we enhanced the phase profiling tool by offering program parallelization hints, which showed the GPU-friendliness indicator for each phase to facilitate the heterogeneous parallel programming, which is described in Section IV-D. In Section IV-E, we evaluate the various design choices of the cache subsystem for the target program as an example to show the capability of design space exploration of our tool.

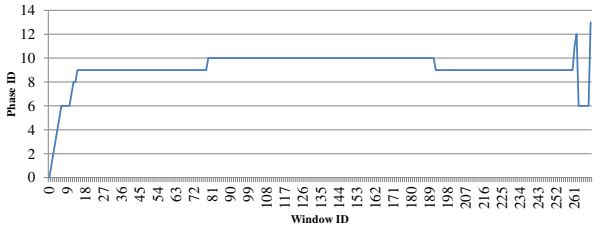

Fig. 5: The phases of the KMP benchmark, where phase ID 9 and 10 are the two phases in the program's main loop.

```
int split(char *txt, auto &strs, char *ch) {
   char *pch = strtok(txt, ch);
   // Loop through all sub-strings
   while (pch != NULL) {
      // Store the string in a vector
      strs.push_back(pch);
      pch = strtok(NULL, ch);
   }
   return strs.size();
}
void create_map(auto &str_map, auto
    &str_vector) {
   // Iterate through str_vector
   for (auto &&v : str_vector) {
      // Accumulate
      str_map[v] += 1;
   }
}
```

Fig. 6: Code for spliting spaces and building a word count map in C++14.

A. Expermental Setup

We used the BananaPi M1 [39] as the real system to validate the results offered by our tool. The real system run Bananian Linux 16.04 [40] with the ARM Cortex-A7 dual-core, 256KB L1 I/D Cache, 1MB L2 Data Cache, global history table branch predictor (256-entry), and 1GB DDR3 Memory. We used *Buildroot* [41] to compile the root filesystem for both real and simulated systems, in order to control the software running on the systems and size of the system image. The timing models of our simulator were calibrated with the hardware specification [42] and related documents [43], [44], [45]. Note that the following experiments used 200k instruction windows for phase analysis and phase threshold was set to 1.

B. Phase Profiling

To showcase the effectiveness of phase-based analysis, we demonstrate an inspiring results from searching algorithm, KMP (Knuth-Morris-Pratt String Matching Algorithm) from MachSuite benchmark [46]. In this case, 1K-instruction windows were used for phase profiling, which is plotted in Fig.5, where the phase ID 9 and 10 represent the respective code segments for the *string searching* and *string matching* within the KMP's main loop. By dividing the number of the execution

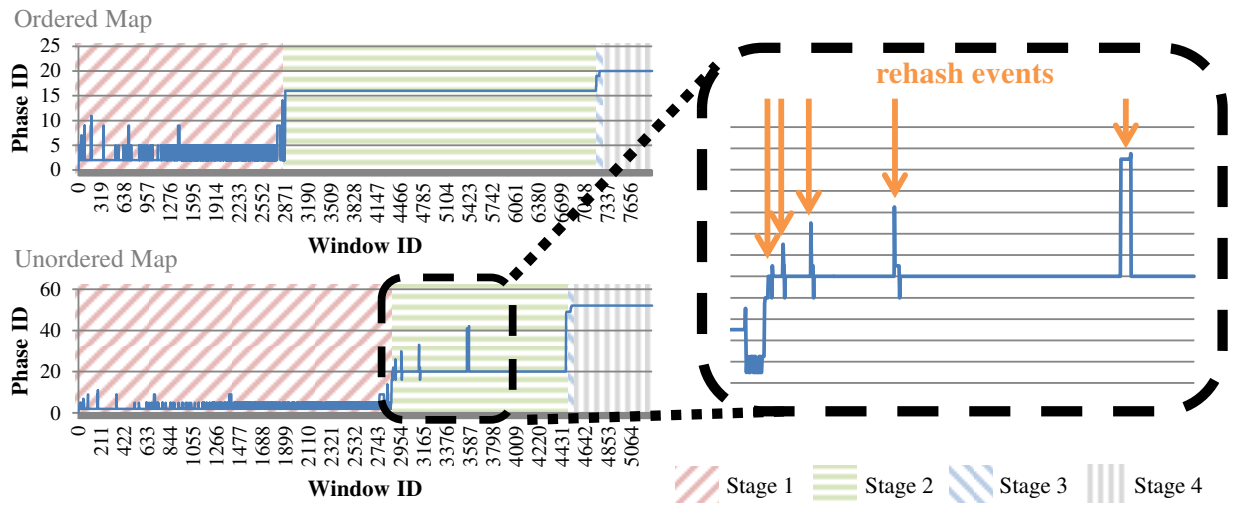

Fig. 7: Phase behaviors of word count with similarity threshold 0.99. Stage 1 is parsing input file, Stage 2 is building word count, Stage 3 is finding the maximum value and Stage 4 is deallocating resources.

count of the matching statement to iteration count, illustrated in our visualization tool, we found an interesting performance difference due to algorithm design and workloads. The results show that the string search codes (phase ID 9) have a probability of almost zero whereas partial matching (phase ID 10) has 5% of probability of finding a matching string in one loop iteration. This experiment demonstrates a way to better understand the runtime behavior of the program for the specific input data, which is hardly offered by the conventional tracing tools.

C. Case study with Word Count

To demonstrate the benefits of phase based profiling, we implemented a word count application which builds a whole dictionary with counters and outputs the word with the maximum number. We use only the first million words from [47] to show the idea of phase detection since more words do not change the trend of results much. In our case study, we implemented the algorithm in *C++14* with *auto type deduction* to be able to use two different data types to build the table, *ordered map* and *unordered map*. The implementation of word count including four stages, (1) reading the text input and parse it into arrays, (2) building the dictionary data table with a accumulated counter, (3) looping through the dictionary to find the maximum counted number and the corresponding word, and lastly, (4) deallocating all resources and memory. The codes of previous two stages are listed in Fig.6.

1) **Performance on Real Platforms:** We ran the applications on BananaPi to test the performance and validate the accuracy of the simulation. On the real platform, the execution time was 2.42s and 4.56s for unordered and ordered map, respectively. The simulation reported estimated execution time of 2.64s and 4.29s for unordered and ordered map, respectively.
2) **Ordered Map Behavior:** With the proposed phase based profiling mechanism, the behavior are presented as four stages of execution in Fig.7. The last stage of execution was deallocating resources happened just before the exit of the main function. In stage 2, the program behavior is very stable and stay in Phase ID 16 for the whole time. In *C++14 STL*, *ordered map* is implemented as a balanced tree structure to maintain the order in $O(log\ n)$ time complexity for every insertion and search. The balanced tree tends to have better performance on range searching since the data is stored in order. However, the cost of maintaining a balanced tree and comparing two keys are known as possible performance issues. In the experimental results, the phase ID 16 showed that the balanced tree implementation is a regular in the whole process.
3) **Unordered Map Behavior:** In Fig.7, the program behavior is also perfectly mapped into four stages of execution. However, unlike *ordered map*, the *unordered map* uses a hash table to implement the data structure. Marked with arrows in Fig.7, *rehash* happened several times in the stage 2 of execution. A *rehash* is usually a painful function to the hash table since it requires allocating new memory region and redo the hash functions to all the keys that have been stored. Depending on applications, the cost of *rehash* might pull down the performance of a hash table. From our phase timeline, we can easily find that the number of *rehash* happens about 5 times without knowing where to instrument probes of events in advance. To improve the performance, one can use a larger size of newly allocated memories for all *rehash* functions to lower the frequency of rehashing. In a large program, the important events are usually not trivial as a simple

Metrics	Data Structure	CPU	Cache			Memory	Branch
			L1I	L1D	L2D		
Miss Rate	Ordered Map	-	0.000	0.019	0.247	-	0.100
	Unordered Map	-	0.000	0.036	0.305	-	0.089
Access Times	Ordered Map	863.04M	231.36M	491.50M	9.92M	2.45M	194.30M
	Unordered Map	310.33M	70.57M	81.46M	3.38M	1.03M	44.34M
Estimated Time(s)	Ordered Map	0.174s	0.790s			0.270s	0.325s
	Unordered Map	0.126s	0.230s			0.190s	0.066s

Table I: Performance counter comparison of the Phase ID 16 (Ordered Map) and Phase ID 20 (Unordered Map).

rehash and thus our proposed mechanism could help shorten the development time greatly since phase based profiling reports the events that matter with collected performance information.

4) **Performance Bottlenecks:** To study the cause of performance difference of two implementations of dictionary data structure, we dig into the performance information of Phase ID 16 from *ordered map* and Phase ID 20 from *unordered map* in Fig.7. Listed in Table I, breaking down the execution time of these two phases, we noticed a serious problem on *ordered map*. Though *ordered map* has lower cache miss rate, the access count of all caches is several times more than the *unordered map*. Even worse, with higher the branch miss prediction rate, the branch count was 4.4x more than the *unordered map*. Despite hash table needs *rehash*, it was still much faster than doing balanced tree due to the nature of hash algorithm requiring much fewer comparison operations and fewer branches on searching. However, it comes with the price of higher data cache miss rate which pollutes the caches and might cause performance issues on other programs.

Usually, one needs to instrument some probes, e.g. *PAPI* [14], or use tracing tools, e.g. *systemtap* [48], in order to collect performance information of events. With the phase detection mechanism, the behavior of a program execution is exploited easily without instrumenting probes. In addition, *Snippit* can help developers to evaluate the performance difference of self-tuning algorithms or changes of data-placement by digging into the phase changes and their performance data.

D. Hint for Program Parallelization

Our framework offers the *GPU-friendliness* indicator to facilitate program parallelization with GPU accelerators. In particular, the tool is able to make a suggestion: which code segments are suitable for the GPU acceleration, by combining the program phases and the data from the simulated PCM. The proposed framework incorporates with the design proposed by Ioana *et al.* [49], which showed the possibility of predicting the magnitude of speedup of a program running on GPU from the performance counters collected from CPU. Furthermore, SVM [50] is adopted as the machine learning engine to find out which phases are seemed to be good candidates for the acceleration, where the number of *total instructions* and the number of instructions of *ALU*, *LD* and *BR* are used as the input features for training the learning model.

Accuracy	Recall	Precision
90.556% (337/371)	87.0504% (121/139)	87.6812% (121/138)

Table II: GPU friendliness prediction result using rodinia benchmark and SVM with 10-fold cross validation.

Model Building. We used Rodinia benchmark suite [51], a representative benchmark for heterogeneous accelerations, to train the machine learning model. Nineteen OpenCL-based programs were selected from the benchmark suite and were run on the evaluation platform, which was a desktop computer equipped with Intel CPU i5-4790, 16GB DD3 Memory, and Nvidia GPU GTX780, where the OpenCL kernels are accelerated by the GTX 780. We compared the delivered performance of the OpenCL and non-OpenCL (serial) versions, and labeled the OpenCL kernels (accelerated regions) as GPU-friendly if the parallel version is 2x faster than the serial version; otherwise, they were labeled as non-GPU-friendly. Besides, the non-OpenCL-kernel codes are marked as non-GPU-friendly. In sum, there were 371 training data, consisting of 139 GPU-friendly data and 232 GPU-non-friendly data in the data set.

GPU-Friendliness Prediction. We used the F-measurement [52] as the metric to indicate the performance of the built model. As listed in Table II, the results show that utilizing phases as the basic units of recording performance counter events achieves 90.56% of accuracy and outperformed the results from the referenced design that was 80% [49]. *Precision* shows that 87.68% of the phases which was predicted as GPU-friendly were actually GPU-friendly, while *Recall* shows that 87.05% of the total number of actually GPU-friendly phases were found.

GPU-Friendliness for Word Count. Our tool was used to provide the GPU-friendliness indicator for the unordered map version of word count. The tool found that 37 out of 56 phases were predicted as GPU-friendly. Breaking down the results into the four program stages, as shown in Fig.7, the ratio of phases predicted as GPU-friendly was 3/15 in stage 1 (splitting data into vectors), 31/32 in stage 2 (building table), 3/3 in stage 3 (find the maximum value and its corresponding word) and 0/6 in stage 4 (deallocating resources). The percentage of GPU-friendliness for each phase indicated the degree of the confidence for GPU acceleration. The percentage numbers of the four stages showed that the proposed tool is very informative as the parallelization suggestions, since all data-parallel parts of the program were found, i.e., stage 2 and 3. It is interesting to note that the three phases, which are

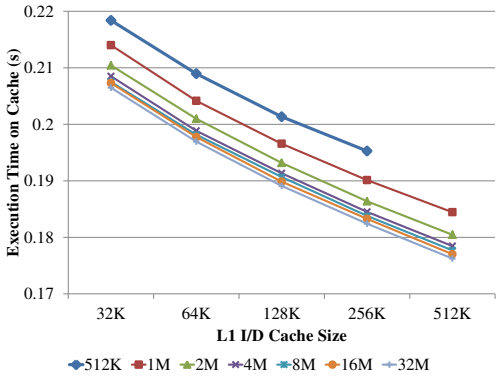

Fig. 8: Execution time spent on cache of Unordered Map (Phase ID 20) when vary L2 cache size with similarity threshold 0.99.

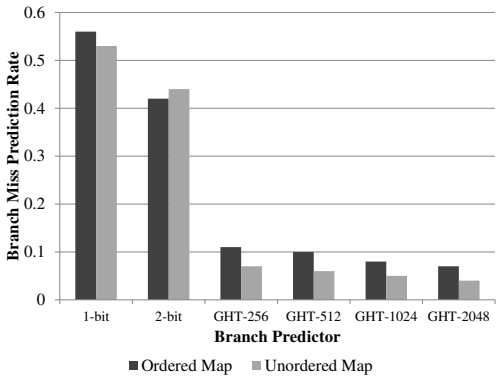

Fig. 9: Branch miss rate comparison of whole program.

recognized as GPU-friendly in stage 1, were doing *resize* to the *word vector* while the rests were having too many branches in a phase which results in GPU-non-friendly. The information helps programmers understand the promising code segments for parallelization before the program is parallelized.

E. Exploring Hardware Parameters

Snippit provides the ability to simulate multiple hardware configurations in one run. With reduced amount of time, developers can find the effectiveness of various hardware configuration points. Most importantly, the performance evaluation is done with the program behaviors, which might be critical algorithm path in the design.

To demonstrate this idea, we altered the L1/L2 cache sizes and evaluated the resulting performance with the timing simulator. Fig.8 plots the delivered performance of different sizes for the caches, 32KB to 512KB for L1 caches and 512KB to 32MB for the L2 cache. The figure shows that increasing the size of either L1 or L2 cache can help the performance. Furthermore, considering the cost and the diminishing returns, the best L2 cache size for word count could be 2 MB and L1 is 256 KB. The experiments demonstrate that the tool is very helpful for both application/system developers to dig into the performance and resolve the bottlenecks.

In addition, we tried to explore different branch predictor for better performance on this application. As shown in Fig.9, the decrease of branch miss rate of Global History Table (GHT) was significantly better than purely 2-bit or 1-bit. However, the benefit of increasing the size of GHT table did not give much improvement while increasing the cost of hardware. These results showed us the cost of the size of branch predictor can be saved while the performance impact on this application was negligible.

V. Related Work

In this section, we review the previous works in these two categories: phase analysis techniques and heterogeneous system simulators. We show that our work is different from the previous works.

Phase Analysis Techniques and Applications. The studies of program phase analysis [53], [25], [27], [54], [27] show that the way a program's execution changes over time is not random but is often structured into repeating behaviors, called program phases. In particular, Timothy *et al.* [55] showed an example of exploiting program phases on SPEC 2000 benchmarks. By using BBVs, they found that while program behavior changes significantly over time, the behavior of all of the performance metrics tends to change in unison, i.e., hardware performance counters tend to be similar in BBVs of the same phase. In an extension of their findings, they proposed SimPoint [54] to accelerate cycle-level simulation by simulating only the representative BBV in each phase. Furthermore, the summarized program behaviors across execution time are used as an important information for guiding the optimizations [30], [29], [56], [57], [58]. For example, these works leverage phase information for guiding cache resizing and dynamic voltage/frequency scaling [29], [56]. There are some works scheduling the threads onto the proper heterogeneous cores according to the program phase information [57], [58]. ScarPhase [27] is a phase profiling tool which estimates program's CPI based on its online phase detection algorithm. It relies heavily on the Intel PEBS for precisely sampling branches to construct the phase data that is similar with the BBV. Without the Intel PEBS support the accuracy of the collected data is low.

Heterogeneous System Simulators. To facilitate the design of heterogeneous systems, many simulators have been developed to assist the performance estimation of the heterogeneous systems [17], [59], [60], [61], [62], [34]. MCEmu [17] provides a simulation framework that is capable of modeling both the ARM-based application processor and the digital signal processing cores in the PAC Duo system-on-chip. In addition, simulators have been developed to help the design space exploration of accelerator-rich platforms [59], [61], [62]. gem5-gpu [60] incorporates the CPU simulator (gem5 [10]) with the GPU simulator (GPGPU-Sim [63]), and the authors claim that gem5-gpu is able to run the unmodified CUDA programs. Lai *et al.* [34] built the timing approximate full system simulator

that integrates GPU simulator from Multi2Sim framework. The resulting heterogeneous system simulator is able to emulate the shared-memory system, where the last-level cache is shared between the CPU and AMD Southern Islands GPU. OpenCL programs are allowed to run on the heterogeneous simulator to evaluate the cache/memory performance and detect race conditions occurred on the programs.

Summary. To the best of our knowledge, we are the first work that integrates the program phase information with the timing approximate simulators to form the simulator-based profiling tool and to guide the heterogeneous system design. In particular, we are not aware of any other heterogeneous system simulator that provides high simulation speed as timing approximate simulators and offers phase-based performance profiles for emulated applications.

On the other hand, while our work adopts the online phase detection algorithm in ScarPhase [27], our work is very different from it. As illustrated in Fig. 4, our profiling tool links the program phase information to the source level which is very helpful for system/application developers to understand runtime activities. In addition, our work does not limited by hardware; that is, our work relies on the counter data collected during the CPU simulation (this can be extended to any architecture supported by QEMU), whereas ScarPhase relies heavily on the Intel's PEBS on the real machine. We believe that our work helps facilitate understanding of the complex software running on the heterogeneous system.

VI. Conclusion

Targeting heterogeneous system and application designer in early stage development, we implemented *Snippit* as a SW/HW co-design tool to accelerate the process of timing simulation for both application profiling and design space exploration. Phase-based profiling provides meaningful information and highlights the important phases to *Snippit* users, allowing the users to find the bottlenecks in time-varying behaviors programs. With the source code mapping to the phases, users can view the performance results associated with a specific line of code and vice versa. In our experiments, we are able to analyze the traversing times of the hash table offered by the standard C++14 library without instrumenting any probe and looking into source code of the libraries. Last but not least, *Snippit* provides insights for further optimizations and predicts the performance for the optimizations on a virtual platform; i.e., the results of GPU-friendliness prediction presents evidence on predicting/finding the codes which are possible to be accelerated or optimized. The results show that *Snippit* helps facilitate the hardware/software co-design process.

Acknowledgments

This work was financially supported by Ministry of Science and Technology of Taiwan under grants MOST No. 105-2221-E-002 -143 -MY2 and No. 106-3114-E-002 -008.

References

[1] C. Stoif, M. Schoeberl, B. Liccardi, and J. Haase, "Hardware synchronization for embedded multi-core processors," in *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*. IEEE, 2011, pp. 2557–2560.

[2] A. Pullini, F. Conti, D. Rossi, I. Loi, M. Gautschi, and L. Benini, "A heterogeneous multi-core system-on-chip for energy efficient brain inspired vision," in *Circuits and Systems (ISCAS), 2016 IEEE International Symposium on*. IEEE, 2016, pp. 2910–2910.

[3] T. D. Han and T. S. Abdelrahman, "Reducing branch divergence in gpu programs," in *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*. ACM, 2011, p. 3.

[4] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.

[5] L. Zhuo, G. R. Morris, and V. K. Prasanna, "High-performance reduction circuits using deeply pipelined operators on fpgas," *IEEE Transactions on Parallel and Distributed Systems*, vol. 18, no. 10, 2007.

[6] D. J. Abadi, P. A. Boncz, and S. Harizopoulos, "Column-oriented database systems," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1664–1665, 2009.

[7] T. Harter, D. Borthakur, S. Dong, A. S. Aiyer, L. Tang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Analysis of hdfs under hbase: a facebook messages case study." in *FAST*, vol. 14, 2014, p. 12th.

[8] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean, "Device placement optimization with reinforcement learning," *arXiv preprint arXiv:1706.04972*, 2017.

[9] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," *arXiv preprint arXiv:1712.01208*, 2017.

[10] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[11] S. L. Graham, P. B. Kessler, and M. K. Mckusick, "Gprof: A call graph execution profiler," in *ACM Sigplan Notices*, vol. 17, no. 6. ACM, 1982, pp. 120–126.

[12] T. Beauchamp and D. Weston, "Dtrace: The reverse engineers unexpected swiss army knife," *Blackhat Europe*, 2008.

[13] A. C. de Melo, "The new linuxperftools," in *Slides from Linux Kongress*, vol. 18, 2010.

[14] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *Proceedings of the department of defense HPCMP users group conference*, vol. 710, 1999.

[15] S. Wallace and K. Hazelwood, "Superpin: Parallelizing dynamic instrumentation for real-time performance," in *Proceedings of the International Symposium on Code Generation and Optimization*. IEEE Computer Society, 2007, pp. 209–220.

[16] M. T. Yourst, "Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator," in *Performance Analysis of Systems & Software, 2007. ISPASS 2007. IEEE International Symposium on*. IEEE, 2007, pp. 23–34.

[17] C. Tu, S. Hung, and T. Tsai, "Mcemu: A framework for software development and performance analysis of multicore systems," *ACM Trans. Design Autom. Electr. Syst.*, vol. 17, no. 4, p. 36, 2012. [Online]. Available: http://doi.acm.org/10.1145/2348839.2348840

[18] D. Sanchez and C. Kozyrakis, "Zsim: fast and accurate microarchitectural simulation of thousand-core systems," in *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3. ACM, 2013, pp. 475–486.

[19] T. E. Carlson, W. Heirmant, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. IEEE, 2011, pp. 1–12.

[20] F. Bellard, "Qemu, a fast and portable dynamic translator." in *USENIX Annual Technical Conference, FREENIX Track*, 2005, pp. 41–46.

[21] S.-H. Hung, T.-W. Kuo, C.-S. Shih, and C.-H. Tu, "System-wide profiling and optimization with virtual machines," in *Design Automation Conference (ASP-DAC), 2012 17th Asia and South Pacific*. IEEE, 2012, pp. 395–400.

[22] C.-K. Lai, C.-W. Yeh, and S.-H. Hung, "Fast race detection and profiling framework for heterogeneous system," in *Computer Symposium (ICS), 2016 International*. IEEE, 2016, pp. 525–530.

[23] R. Ubal, J. Sahuquillo, S. Petit, and P. López, "Multi2sim: A simulation framework to evaluate multicore-multithread processors," in *IEEE 19th International Symposium on Computer Architecture and High Performance computing, page (s)*. Citeseer, 2007, pp. 62–68.

[24] T. Sherwood, E. Perelman, and B. Calder, "Basic block distribution analysis to find periodic behavior and simulation points in applications," in *Parallel Architectures and Compilation Techniques, 2001. Proceedings. 2001 International Conference on*. IEEE, 2001, pp. 3–14.

[25] A. Sembrant, D. Black-Schaffer, and E. Hagersten, "Phase behavior in serial and parallel applications," in *Workload Characterization (IISWC), 2012 IEEE International Symposium on*. IEEE, 2012, pp. 47–58.

[26] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *ACM SIGARCH Computer Architecture News*, vol. 30, no. 5. ACM, 2002, pp. 45–57.

[27] A. Sembrant, D. Eklov, and E. Hagersten, "Efficient software-based online phase classification," in *2011 IEEE International Symposium on Workload Characterization (IISWC)*, Nov 2011, pp. 104–115.

[28] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*. ACM, 2000, pp. 245–257.

[29] A. S. Dhodapkar and J. E. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*. IEEE, 2002, pp. 233–244.

[30] J. E. Smith and A. S. Dhodapkar, "Dynamic microarchitecture adaptation via co-designed virtual machines," in *Solid-State Circuits Conference, 2002. Digest of Technical Papers. ISSCC. 2002 IEEE International*, vol. 1. IEEE, 2002, pp. 198–199.

[31] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *ACM SIGARCH Computer Architecture News*, vol. 31, no. 2. ACM, 2003, pp. 336–349.

[32] P. E. Black, "Manhattan distance," *Dictionary of Algorithms and Data Structures*, vol. 18, p. 2012, 2006.

[33] J. Burkardt, "K-means clustering," *Virginia Tech, Advanced Research Computing, Interdisciplinary Center for Applied Mathematics*, 2009.

[34] C. Lai, C. W. Yeh, C. Tu, and S. Hung, "Fast profiling framework and race detection for heterogeneous system," *Journal of Systems Architecture - Embedded Systems Design*, vol. 81, pp. 83–91, 2017. [Online]. Available: https://doi.org/10.1016/j.sysarc.2017.10.010

[35] 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.

[36] S.-H. Hung, F.-T. Liang, C.-H. Tu, and N. Chang, "Performance and power estimation for mobile-cloud applications on virtualized platforms," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2013 Seventh International Conference on*. IEEE, 2013, pp. 260–267.

[37] T.-H. Chiang, C.-Y. Liu, C.-W. Yeh, C.-H. Tu, and S.-H. Hung, "Program analysis with a loop-function-based tracing tool on virtual platforms," in *Proceedings of the International Conference on Research in Adaptive and Convergent Systems*. ACM, 2017, pp. 255–260.

[38] A. S. Dhodapkar and J. E. Smith, "Comparing program phase detection techniques," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003, p. 217.

[39] F. Sinvoid, "Bananapi m1," http://www.banana-pi.org/m1.html.

[40] "Bananian linux 16.04," https://www.bananian.org/news#bananian_linux_1604_released_-_2016-04-23.

[41] E. Andersen, "Buildroot: making embedded linux easy," https://buildroot.org/.

[42] M. Rullgard, "Cortex-a7 instruction cycle timings," http://hardwarebug.org/2014/05/15/cortex-a7-instruction-cycle-timings/.

[43] M. Pricopi, T. S. Muthukaruppan, V. Venkataramani, T. Mitra, and S. Vishin, "Power-performance modeling on asymmetric multi-cores," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on*. IEEE, 2013, pp. 1–10.

[44] W.-C. Hsu, S.-H. Hung, and C.-H. Tu, "A virtual timing device for program performance analysis," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*. IEEE, 2010, pp. 2255–2260.

[45] S.-h. Kang, D. Yoo, and S. Ha, "Tqsim: A fast cycle-approximate processor simulator based on qemu," *Journal of Systems Architecture*, vol. 66, pp. 33–47, 2016.

[46] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *Workload Characterization (IISWC), 2014 IEEE International Symposium on*. IEEE, 2014, pp. 110–119.

[47] M. Mahoney, "Testset from data compression program benchmark and word2vec," http://mattmahoney.net/dc/text8.zip.

[48] F. C. Eigler and R. Hat, "Problem solving with systemtap," in *Proc. of the Ottawa Linux Symposium*. Citeseer, 2006, pp. 261–268.

[49] I. Baldini, S. J. Fink, and E. Altman, "Predicting gpu performance from cpu runs using machine learning," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*. IEEE, 2014, pp. 254–261.

[50] C.-C. Chang and C.-J. Lin, "Libsvm: a library for support vector machines," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 2, no. 3, p. 27, 2011.

[51] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee, 2009, pp. 44–54.

[52] D. M. Powers, "Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation," 2011.

[53] M. Van Biesbrouck, T. Sherwood, and B. Calder, "A co-phase matrix to guide simultaneous multithreading simulation," in *Performance Analysis of Systems and Software, 2004 IEEE International Symposium on-ISPASS*. IEEE, 2004, pp. 45–56.

[54] B. Calder, T. Sherwood, G. Hamerly, and E. Perelman, "Simpoint: Picking representative samples to guide simulation," *Performance Evaluation and Benchmarking*, p. 117, 2005.

[55] T. Sherwood, E. Perelman, G. Hamerly, S. Sair, and B. Calder, "Discovering and exploiting program phases," *IEEE micro*, vol. 23, no. 6, pp. 84–93, 2003.

[56] K. Meng, R. Joseph, R. P. Dick, and L. Shang, "Multi-optimization power management for chip multiprocessors," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 177–186.

[57] L. Sawalha, S. Wolff, M. P. Tull, and R. D. Barnes, "Phase-guided scheduling on single-isa heterogeneous multicore processors," in *Digital System Design (DSD), 2011 14th Euromicro Conference on*. IEEE, 2011, pp. 736–745.

[58] T. Sondag and H. Rajan, "Phase-based tuning for better utilization of performance-asymmetric multicore processors," in *Code Generation and Optimization (CGO), 2011 9th Annual IEEE/ACM International Symposium on*. IEEE, 2011, pp. 11–20.

[59] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures," in *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*. IEEE, 2014, pp. 97–108.

[60] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood, "gem5-gpu: A heterogeneous cpu-gpu simulator," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 34–36, 2015.

[61] J. Cong, Z. Fang, M. Gill, and G. Reinman, "Parade: A cycle-accurate full-system simulation platform for accelerator-rich architectural design and exploration," in *Computer-Aided Design (ICCAD), 2015 IEEE/ACM International Conference on*. IEEE, 2015, pp. 380–387.

[62] Y. S. Shao, S. L. Xi, V. Srinivasan, G.-Y. Wei, and D. Brooks, "Co-designing accelerators and soc interfaces using gem5-aladdin," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*. IEEE, 2016, pp. 1–12.

[63] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing cuda workloads using a detailed gpu simulator," in *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*. IEEE, 2009, pp. 163–174.