Spectral Profiling: Observer-Effect-Free Profiling by Monitoring EM Emanations

Nader Sehatbakhsh*, Alireza Nazari†, Alenka Zajic‡ and Milos Prvulovic§
Georgia Institute of Technology
Atlanta, Georgia
Email: *nader.sb@gatech.edu, †anazari@gatech.edu, ‡alenka.zajic@ece.gatech.edu, §milos@cc.gatech.edu

***Abstract*—This paper presents Spectral Profiling, a new method for profiling program execution without instrumenting or otherwise affecting the profiled system. Spectral Profiling monitors EM emanations unintentionally produced by the profiled system, looking for spectral "spikes" produced by periodic program activity (e.g. loops). This allows Spectral Profiling to determine which parts of the program have executed at what time. By analyzing the frequency and shape of the spectral "spike", Spectral Profiling can obtain additional information such as the per-iteration execution time of a loop. The key advantage of Spectral Profiling is that it can monitor a system as-is, without program instrumentation, system activity, etc. associated with the profiling itself, i.e. it completely eliminates the "Observer's Effect" and allows profiling of programs whose execution is performance-dependent and/or programs that run on even the simplest embedded systems that have no resources or support for profiling. We evaluate the effectiveness of Spectral Profiling by applying it to several benchmarks from MiBench suite on a real system, and also on a cycle-accurate simulator. Our results confirm that Spectral Profiling yields useful information about the runtime behavior of a program, allowing Spectral Profiling to be used for profiling in systems where profiling infrastructure is not available, or where profiling overheads may perturb the results too much ("Observer's Effect").**

I. Introduction

Profiling of program execution is an essential part of performance optimization and performance analysis efforts. Typically, the goal of profiling is to identify the regions of code where the bulk of the execution time is spent ("hot" regions), which allows developers or the compiler to focus their optimization efforts. Another common goal of profiling is to gain more insight into the performance characteristics of some region of code, such as typical execution time or the variation in execution time for a code fragment, which can help programmers understand a performance problem and identify potential solutions [1], [2], [3], [4].

In practice, profiling is typically implemented through adding instrumentation to the program. At runtime, this instrumentation updates statistics about program execution at runtime, e.g. by counting how many times a particular point in the code was encountered during execution[1], [3], [5], [6]. For hot-region profiling, the alternative to instrumentation is to periodically interrupt execution and sample the program counter [2], [7], [8], [9], [10]. In recent years, support for profiling has been added to the hardware, especially for higher-end processors, which reduces (but does not eliminate) the execution time overheads (and the "Observer's Effect") of profiling, while increasing the cost of the processor even when profiling is not needed [8], [11], [12], [13], [14].

For some systems, especially in real-time and cyber-physical domains, program execution can change significantly when instrumentation (or profiling-induced interrupts) change the performance characteristics of the program. For example, these programs often include different algorithms that are used when there is a risk of not meeting real-time deadlines or safety criteria, and profiling instrumentation and/or interrupts may cause the system to use these "emergency" algorithms more often, leading to an execution profile that is no longer representative of profiling-free execution.

Profiling is also difficult in systems that have very limited resources, e.g. the processor performance and/or memory capacity may be barely sufficient for the system's primary function and cannot accommodate profiling overheads, the system's power source (e.g. energy harvesting) may not be able to support the added energy consumption to collect, store, process, or transmit the profiling data, and the processor has no advanced hardware support for profiling because it would significantly add to its (extremely low) cost.

Unfortunately, many Internet-of-things (IoT) devices have both types of limitations - real-time/cyber-physical system operating under severe cost, energy, etc. limitations.

Even in some resource-rich profiling scenarios, e.g. already-deployed systems that suffer from unexplained performance problems, it would be highly desirable to profile the program execution as-is, without changing the program or the system activity in any way, to ensure that actual program behavior in that deployment is captured during profiling.

In this paper we present *Spectral Profiling*, a new approach to profiling that leverages unintentional EM emanations from the profiled systems. Spectral Profiling allows highly accurate profiling of loops and other repetitive activity, without perturbing the profiled system, the program it runs, or the characteristics of the execution, in any way.

The key insight in spectral profiling is that repetitive program activity (e.g. a loop) causes the unintentional EM signals to exhibit periodicity, i.e. the spectrum of these EM signals will have "spikes" at frequencies that correspond to the time spent

*,† These two authors contributed to the paper similarly (author order does not reflect the extent of contribution).

978-1-5090-3508-3/16/$31.00

in each repetition of the program activity. For example, a loop whose per-iteration time is T will create a spike in the EM spectrum at frequency $f = 1/T$ and multiples (harmonics) of that frequency. Spectral profiling relies on training with known inputs to identify the spike frequencies that are characteristic for each loop. During profiling, Spectral Profiling monitors the spectrum in real time to identify when spikes characteristic for each loop appear and disappear, allowing it to determine when each loop is entered and exited. Additionally, the frequency of the spike and its shape allow Spectral Profiling to determine the average per-iteration time for each loop and the distribution of the per-iteration time around that average.

The main contributions of this paper are:

- A new approach for profiling that requires no profiling-related support or activity on the profiled system.
- A proof-of-concept implementation of this approach to demonstrate its feasibility in practice.
- An experimental evaluation that shows our new approach achieves high profiling accuracy, both in a real system and in cycle-accurate simulation.

The remainder of this paper is organized as follows. Section II presents the background related to our new approach. Section III describes the Spectral Profiling approach and details our proof-of-concept implementation. Section IV presents the evaluation setup and results for our proof-of-concept implementation of Spectral Profiling and some analysis about the runtime behavior of loops and how some architectural parameters of the profiled system affect our results. Finally, Section V briefly reviews related work and Section VI presents our overall conclusions.

II. Background

Electronic circuits within computers generate EM emanations [15], [16], [17] as a side-effect of current flows. Since current flows in the systems can vary with program activity, these EM emanations often convey important information about program activity in the system. Most research work on EM emanations has focused on the risks they create as a side-channel, i.e. as a way for attackers to extract sensitive data values (such as cryptographic keys) from the system [16], [18], and on countermeasures against such attacks, primarily for smart-cards used for authentication and payments [19], [20], [21], [22], [23], [24], [25], [26], [27], [28].

Beyond extracting sensitive data values, side-channel emanations have also been used to learn more about program behavior, e.g. to identify webpages during browsing [29] or find anomalies in software activity [30], [31] without attributing emanations to specific parts of the code. However, results in [32], [33], [34], [35], [36] show that differences between different instructions can be measured in EM analog signals across different devices (e.g. desktops, laptops, FPGAs), identify which aspects of program activity modulate which EM-emanated signals [34], and even attribute emanations to specific dynamic paths in the code [36].

The goal of this paper is to leverage unintentional EM emanations for *efficient, accurate, and observer-effect-free* software profiling that can be applied to realistic program runs, where many millions of basic blocks are executed per second. This requires understanding the properties of EM signals created by typical program behaviors (loops and other repetitive activity), and exploiting these properties to efficiently match the EM signal to program code in large (e.g. million-cycle) chunks.

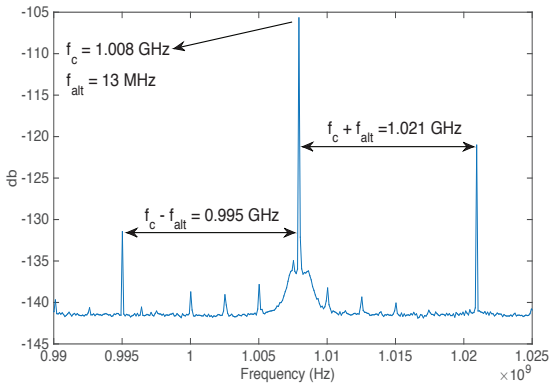

Fig. 1. Spectrum of an AM modulated loop activity.

To illustrate main idea behind spectral profiling, Fig. 1 shows a spectrum of an AM modulated loop activity. We can observe a spike in a spectrum at $f_{clock} = 1.0079$ GHZ. This signal is created by the processor clock periodic activity and acts as a carrier signal in AM modulation. On the left and the right side from the carrier, we can observe two spikes. They correspond to loop activity with execution time of 77 ns ($\approx$ 13 MHz) in "bit_count" benchmark from MiBench [37] suite that is AM modulated onto a carrier clock frequency $f_{clock} = 1.0079$ GHz. Please note that two spikes in the spectrum are an artifact of AM modulation. Fig. 1 indicates that if we observe a program spectra during runtime, we can deduce which spikes correspond to current executing path(s). By knowing the frequency of each path before runtime, during execution we can deduce (1) which path in the program is currently active, (2) how much time program has spent in this path, and (3) how many times this path has been executed.

As an application goes through different functions and loops, its spectrum is expected to change over time. To capture this dynamic spectrum behavior of an application, we use a short-time Fourier transform (STFT) that is defined as [38]

$$STFT\{s(t)\}(\tau,\omega) \equiv X(\tau,\omega) \quad (1)$$

$$= \int_{-\infty}^{+\infty} \mathrm{x(t)w(t-\tau)e}^{-j\omega t} dt, \quad (2)$$

and then compute a spectrogram as follows:

$$spectrogram(t,w) = |STFT(t,w)|^2 \quad (3)$$

In STFT, a long signal is divided into shorter, equal, and slightly overlapping segments (windows). Short windows yield better time resolution, but low frequency resolution, than longer windows, and overlapped windows (typically 50% to

100% overlap) are often used to avoid missing something that occurs at the transition between windows. In the next section we describe methodology of proposed Spectral Profiling.

III. Spectral Profiling

This section describes the Spectral Profiling approach, and our current implementation of it, in more detail. To illustrate how Spectral Profiling works, we use the "Basicmath" application from the MiBench [37] suite as a running example.

Spectral Profiling has two phases, training and profiling. In the training phase, we run the application with known training inputs to identify which spectra correspond to which part of the program (mostly loops), and also to identify the valid orderings between the parts of the program. In the profiling phase, we run the application with unknown inputs, record how the spectrum change over time, and combine that with the information from training to detect which part of the program is executing at each point in time.

Spectral Profiling's recognition of activity is based on recognizing the corresponding spectrum. Any spectrum fundamentally corresponds to the signal observed over some interval of time (window), and the duration of this window represents a trade-off between temporal resolution and frequency resolution. Temporal resolution corresponds to being able to tell where exactly some program activity begins and ends. Fundamentally, a spectrum that corresponds to some time window "blurs together" activity for the entire window, so spectra collected with a short window allow more precise identification of the time when program activity has changed. This means that, to improve temporal resolution, we should use spectra collected over very short intervals of time. However, the number of frequency bins (i.e. the frequency resolution) in the spectrum is proportional to the duration of the time interval, so a spectrum collected over a very brief interval "lumps together" similar frequencies into one frequency bin. This means that two program activities that have spectral "spikes" with different shapes and/or similar frequencies cannot be told apart when using short-window spectra because the spectrum only has one bin for the entire frequency range where both spikes are.

Thus the time window should be short enough to capture relevant events in the profiled execution, e.g. it should be shorter than the duration of most loops - intuitively, attribution of execution time to specific code in the application will be performed at the granularity that is similar to the size of the window. In our *Basicmath* benchmark example, we use a window of 1ms with 75% overlap between consecutive windows, which provides attribution with 0.25 ms granularity and precision between 0.25 ms and 1 ms.

A. Training Phase

The goal of the training phase is to collect spectral signatures for all regions of the program, and also to identify the possible/probable sequence of the program's execution, i.e. when one region of code is executed, which regions can possibly be executed immediately after that.

When we collect spectra during training, we face a dilemma. We can run the program just like we do for profiling (no modifications to the code, no change to the system, no collection of any information on the profiled system). The spectra obtained that way will then be the same as the spectra obtained during profiling, except when spectra produced by a region of code (e.g. a loop) are input-dependent. However, when training spectra are collected this way we do not know which spectrum corresponds to which part of the code.

Alternatively, we can instrument the program (or use interrupt-based sampling) to record which part of the code executes at what time, but the spectra collected in such execution are distorted by such changes and will poorly match the corresponding spectra during profiling.

Our current approach to resolving this dilemma is to first use instrumentation to measure the average per-iteration execution times for each loop. Then we re-run the program with the same training inputs but without the instrumentation to get the undistorted spectra. Finally, we use the per-iteration execution times and the frequencies of spikes in the spectrum to create spectrum-to-loop mappings.

1) Finding Per-Iteration Execution Time: We place instrumentation at the beginning and end of the loop body to get timestamps at those points, compute execution time for each iteration, and store it along with information about which loop they correspond to. When this training run ends, we compute the average per-iteration execution time for each loop instance. Note that this per-iteration time is *only* used in training to identify the frequency at which the corresponding spectrum should have a spike: if per-iteration time of a loop is T, we will expect the corresponding spectrum to have a spike at a frequency that is relatively close to $f = 1/T$.

In our *Basicmath* example, there are four different loops in the source code. Each of the four loops is instrumented to collect per-iteration execution time T, and Fig. 2 shows, for each loop, a histogram of frequencies $f = 1/T$ that correspond to these per-iteration execution times (i.e., 2.(a) corresponds to loop 1, 2.(b) corresponds to loop 2 and so on). This provides us with the approximate frequency at which to expect a spectral "spike" for each loop, along with information about the width and shape of each spike.

2) Finding Spectral Signatures for Each Loop: After calculating the frequency of each loop, we re-run the application with same inputs but without any instrumentation or profiling-related activity on the profiled device, and record the spectra for each time window. In each spectrum, we identify the spikes, then compare their frequencies and shapes to the histogram obtained from the previous (instrumented run). The matches are imperfect because instrumentation perturbs the execution time of a loop's iteration, and thus changes the frequency and shape in the histogram. However, our matching is highly accurate because frequencies that correspond to different loops tend to differ more than the instrumentation-induced errors, because the error introduced by instrumentation is usually in the same direction (increases the per-iteration execution time), and also because our matching approach

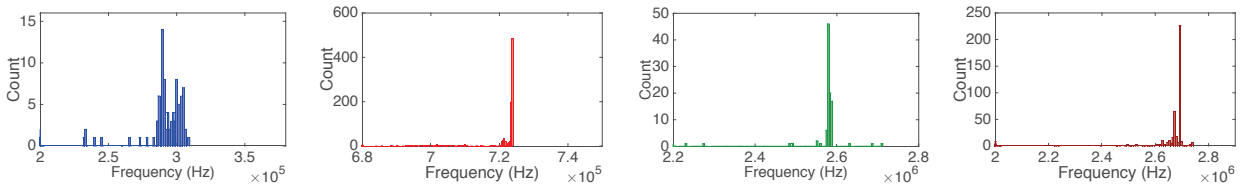

Fig. 2. Histogram of frequencies that correspond to per-iteration execution time ($f = 1/T$) for four different loops in Basicmath benchmark for small (training run) inputs.

TABLE I
MEASURED AND CALCULATED FREQUENCY FOR LOOPS IN "BASICMATH" APPLICATION

Loop Number	Frequency (measured)	Frequency (calculated)
Loop 1	289.12 KHz	289.1 KHz
Loop 2	720.3 KHz	721 KHz
Loop 3	2.628 MHz	2.577 MHz
Loop 4	2.733 MHz	2.69 MHz

utilizes the fact that the two runs used the same inputs and thus have the same sequence of loops. For example, Loop 3 and Loop 4 have relatively similar frequencies, but because we know that Loop 3 is likely to have a lower frequency than, and be executed before, Loop 4, the spectra corresponding to these loops can still be correctly "assigned". In addition, after successfully assigning spectra to the loops, we will also have the sequences of the "assigned" loops.

Table I shows the list of frequencies for four loops in *Basicmath*. The "measured" column in the table shows the actual frequency of the loop (i.e. in the instrumentation-free run), and the "calculated" column shows the average frequency calculated from the instrumentation-enabled histogram. The relative error between the calculated and measured frequency for these loops is up to 2%, but we can still easily match them. Also note that the frequency error introduced by instrumentation increases as the frequency increases. This is because instrumentation has more effect on tight loops (short time per iteration, i.e. high frequency).

After matching spectra to loops, we pre-process the (instrumentation-free) spectrum that corresponds to each loop to identify the "spectral signature" for the loop. In our implementation, the signature is a list of frequencies for the strongest spikes in the spectrum, after removing spikes that appear in all spectra (e.g. for EM signals, for example, this eliminates spikes caused by radio stations, etc.). Note that the signature is not just one number that corresponds to the fundamental frequency of the loop. Some loops have a group of spikes instead of one spike, because their per-iteration execution time takes several discrete values (with some variation around each of them). In most cases, the spectrum also contains not only the spikes that correspond to the per-iteration execution time (fundamental frequency), but also spikes at multiples (harmonics) of that frequency. These additional spikes help differentiate spectra that correspond to different loops, so the signature we use includes all spikes whose magnitude is sufficiently above the noise floor.

B. Profiling Phase

In training, we identified the spectral signature for each loop, and we have also identified the possible/probable sequences of loops (essentially, which loops can execute immediately after which other loops). Profiling consists of running the application with unknown inputs and obtaining profiling information about those runs.

1) Matching of Loop Spectra: Because the profiling inputs are different from training inputs, it is natural to wonder if the spectrum of a loop will change. We have found that many loops, primarily innermost loops, have spectra that are nearly identical to those found in training. Intuitively, the spectrum changes when the per-iteration execution time changes, and in many loops only the number of iterations changes significantly from input to input, but the work of each iteration (and the statistics of branches and architectural events) remain similar. We call these Loops with Input-Independent Spectra (LIIS), and for these loops the spectrum can be matched to the corresponding spectrum from training.

During profiling, we use the same time window we used during training. For each time window during profiling, we obtain the spectrum for that window, identify the spikes in the spectrum (the spectral signature) and compare that signature to the signatures obtained during training. The comparison is performed by attempting to match the peaks in the profile-time and training-time signature. For each peak in the profile-time signature, we find the closest peak (according to frequency) in the training-time signature. If that closest frequency differs too much, the peak remains unmatched. If the closest frequency is very similar, the peak is counted as matched. After attempting to match each peak, the number of successfully matched peaks is used as the similarity metric between the signatures.

If the similarity is high between a profile-time spectral signature and the best-matching training-time signature of a loop, we attribute the execution during that profile-time window to that loop. For the vast majority of time-windows

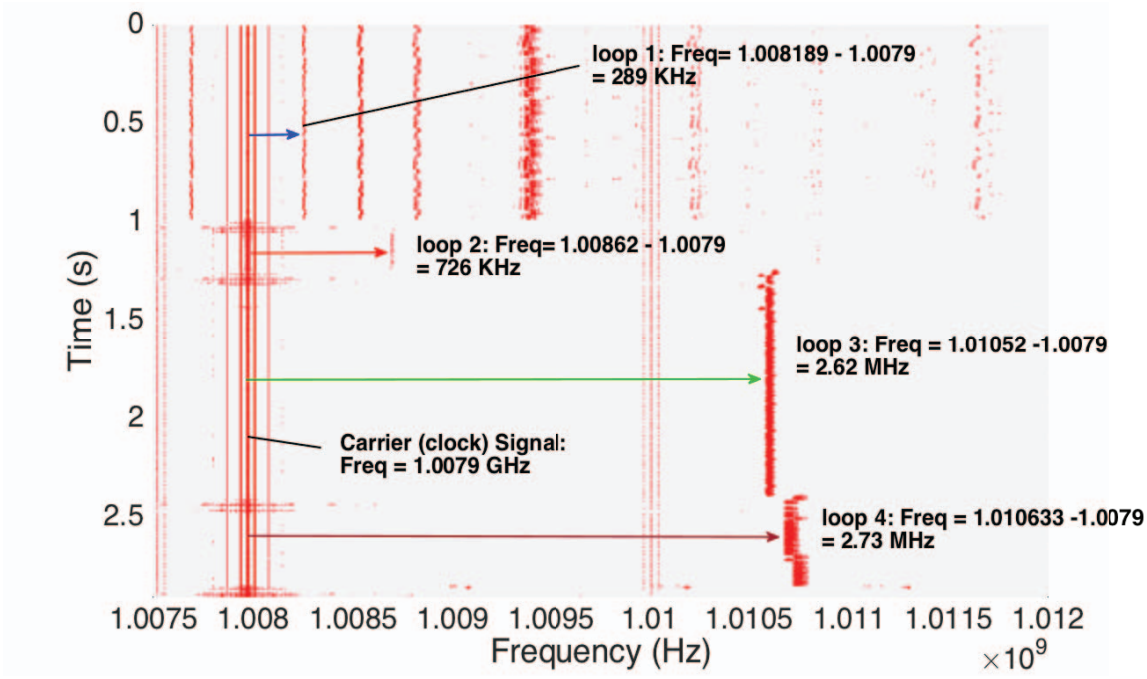

Fig. 3. Spectrogram for *Basicmath* benchmark with large (profiling run) inputs.

that belong to LIIS loops, this similarity is very high and the execution is correctly attributed to the correct LIIS loop.

However, it is possible that none of the profile-time signatures matches the observed signature well enough. This happens primarily because the spectrum of some loops does change with frequency. For example, a command-line flag may cause every iteration of the loop to take one path in one execution and a significantly different path in another execution or a set of control flows inside the loop that can change the per-iteration execution time of the loop. For these loops, the spectrum still indicates that a loop is executing (spikes in the spectrum) and when the loop begins and ends (spikes appear at one time and disappear later) but the spectrum during profiling no longer matches any of the spectra from training.

C. Sequence-Based Matching

To attribute execution time to these Non-LIIS loops (and report their per-iteration execution time during the profiling run), we rely on the model of possible loop-level sequences constructed during profiling. Sequence-based matching begins after LIIS matching is completed for LIIS loops. The spectra from time windows that remain unmatched after LIIS matching are first clustered according to the same similarity metric we used to match LIIS loops to spectra from training, i.e. spectra that have many spikes at similar frequencies will be clustered together. At this point we have clusters where each cluster corresponds to a Non-LIIS loop, but we do not yet know which loop in the code this cluster corresponds to.

However, for each "mystery spectrum" we know that it should be matched to a region of code that is not a LIIS loop, and the LIIS loop spectra observed before and after the "mystery spectrum" tell us which loops have been executed before and after each instance of the "mystery" loop whose cluster we are considering. Fortunately, the model of the application's loop-to-loop transitions restricts the possibilities for matching so that usually only a single Non-LIIS loop remains as a possible match. When there are multiple possible matches, i.e. the "mystery spectra" in a cluster could possibly belong to more than one Non-LIIS loop, we match the cluster to the Non-LIIS loop whose training signature has the highest average similarity to the spectra in the cluster.

Fig. 3 shows the profiling-time spectrogram of the *Basicmath* application. The bold line at 1.008 MHz is the clock signal. The periodic program behavior amplitude-modulates this signal, and the straight lines to the right of this line represent the upper sideband of the modulated signal, i.e. they have the spectrum that corresponds to program behavior but that spectrum is shifted upward in frequency by the clock frequency [34]. In this execution the four loops are executed one after the other (shown by arrows). In this case all four loops were matched through LIIS matching, but if any one of the was not matched through LIS matching, it would still be successfully matched through sequence-based matching.

IV. Results

This section presents our experimental results, first for profiling of execution in real systems through EM emanations, and then for profiling through power signals generated through cycle-accurate simulation. We tested 13 different applications from MiBench [37] benchmark suite on both of these plat-

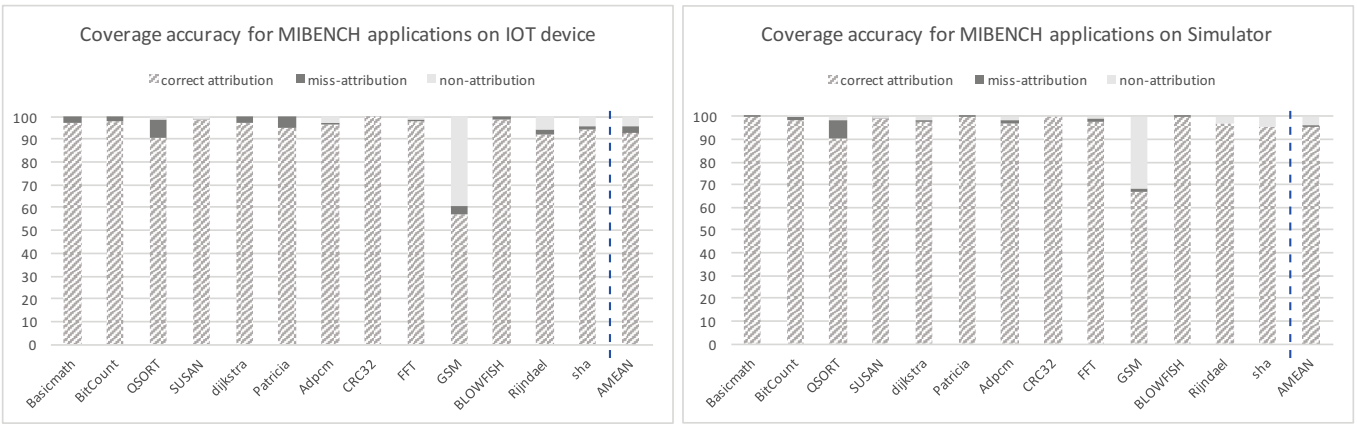

Fig. 4. Correct attribution (striped portion) as a percentage of the overall profiled execution time.

forms. For real system, we used A13-OLinuXino-MICRO [39]. A13-OLinuXino is a single-board Linux computer which has ARM Cortex A8 processor [40]. For cycle-accurate simulator, we used SESC simulator [41].

A. EM-based Spectral Profiling on Real Systems

1) Experimental Setup: To demonstrate the feasibility and effectiveness of Spectral Profiling, we used it to profile applications running on a single-board computer (A13-OLinuXino), which has a 2-issue in-order ARM Cortex A8 processor with 32kB L1 and 256KB L2 caches, and uses Debian Linux as its operating system (OS). Our Spectral Profiling for this system uses electromagnetic (EM) emanations that are received by a commercial small electric antenna (PBS-E1) ([42]) that is placed next to the profiled system's processor. The antenna is placed where the clock signal has the strongest Signal-to-Noise ratio (SNR).

A spectrum analyzer (Agilent MXA N9020A) is then used to record the spectra of the signals collected by the antenna. A spectrum analyzer can be relatively costly (several tens of thousands of dollars), but we elected to use a spectrum analyzer primarily because it provides calibrated measurements, and already has support for automating measurements and for saving and analyzing measured results. In additional experiments, we observed similar spectra with less expensive (<$5,000) commercial software-defined radio receivers.

2) Measurement-Based Results: We apply Spectral Profiling to all 13 applications from the automotive, communications, network, and security categories in the MiBench suite. We used a 1 ms window size with 75% of overlap between windows in all applications except GSM, where we used a 0.5 ms window to improve temporal resolution for attribution of execution time for short-lived loops.

For training, markers are inserted as described in Section 3, except for very tight loops where markers are inserted before and after each loop and, if needed, only iteration-counting is added to the loop. The per-iteration time in this case is computed by dividing the between-markers time by the number of iterations. Each marker reads and records the current clock cycle count from the *ARM Performance Counter Unit* (ARM-PMU)[14], which provides information similar to the x86 "rdtsc" instruction [43]. The training runs are repeated with several different command line flags, in order to identify the sequence of loops that can occur in each application.

The insertion of markers and the identification of possible sequences for an application are both accomplished fully automatically. Identification of loop nests and marker insertion are implemented as a Clang tool, and identification of possible loop sequences in implemented as a pass in LLVM 3.7.

After training, for which we used the small input set [37], we perform actual profiling with the original unmodified code (no markers) and with the large input set [37]. The accuracy we measure is defined as the fraction of execution time for which our method correctly identifies the loop that is currently executing. This accuracy is not 100% because of (1) ***miss-attribution***, during which our algorithm matches the spectrum to a different loop (i.e. loop A is actually executing, but the algorithm matches the spectrum to loop B instead) at loop B, and (2) ***non-attribution***, during which our algorithm finds that the spectrum is too different from loop spectra observed in training, so it leaves such intervals un-attributed. Non-attribution is typically a result of computation whose spectrum varies widely depending on inputs, or activity that has no recognizable spectral signature (e.g. loops whose per-iteration time varies a lot from iteration to iteration).

Fig. 4 (left) shows the breakdown of profiled execution time into time that was accurately attributed, time that was miss-attributed, and non-attributed time. In all benchmarks except GSM, our method provides correct attribution during at least 90% of the execution time, with the arithmetic mean at 93%. Miss-attribution occurs during less than 4% of the execution time, except in QSORT, where miss-attribution occurs during 8% of the time. The larger miss-attribution for QSORT occurs primarily because the std::qsort library function does not have a stable signature, so it is often miss-attributed. It is quite possible that the variation in std::qsort spectra is a result of having multiple loops in that function. Unfortunately, our marker insertion did not include library code so our scheme

treats the entire std::qsort function as a single entity and expects it to have the same spectrum throughout its execution. We expect that this problem can be overcome by also adding loop markers to library code. Overall, the arithmetic mean for miss-recognition is 2.26%.

B. Simulated Results

To provide further evidence that the ability to do Spectral Profiling is a result of a fundamental connection between repetitive program behavior and the spectra of resulting side-channel signals, we apply Spectral Profiling power signals produced via cycle-accurate architectural simulation in SESC [41], a cycle accurate simulator that includes CACTI [44] and WATTCH [45] power models. In this simulation, we model a 1.8GHz 4-issue out-of-order core with 32KB L1 and 64MB L2 caches.

Fig. 4 (right) shows the profiling accuracy results from these simulations, with the accurate-attribution-percentage at 98% on average. This is slightly better than for our real-system results, mainly because simulation-produced power signals are free of radio-frequency noise and other problems that present in EM signals received from real systems are: measurement error, frequency-dependent distortion (for some EM frequencies the real system acts as a better "transmitter" than for others), etc. The arithmetic mean for miss-recognition in simulations is 1.19% which is again slightly better than for the real system.

Overall, Spectral Profiling remains effective in spite of differences in clock rates (1.008 GHz vs. 1.8 GHz), pipelines (In-order vs. Out-of-order), use of different signals (EM emanations vs. power), etc. This provides strong evidence that the existence of spectral signatures is not an anomaly of the particular system we used in our experiment, and that Spectral Profiling is likely to be effective for a wide variety of other real computer systems.

C. Loops with Input-independent Spectra

As discussed in previous section, some loops produce spectral "spikes" whose frequency does not change significantly with changing inputs, while others have input-dependent spectra. Recall that the frequencies of the "spikes" depend on the loop's average per-iteration time and not on the number or iteration executed in the loop, so loops with input-independent spectra (which we abbreviate as LIIS) tend to be innermost loops. Conversely, loops with input-dependent spectra tend to be outer loops whose inner loops have input-dependent iteration counts, or loops that have a set of control flows which can change the per-iteration execution time of the loop. The reason we are interested in LIIS loops is that, during profiling, their execution can be directly recognized from the spectrum. The remaining loops may still be correctly attributed, but that attribution consists of (1) identifying stable spectral patterns (the spectrum has "spikes" that remain the same for a while, which indicates that the system is likely executing a single loop) and (2) using the program's possible loop-level sequences (learned from training), i.e. the knowledge of which loops may possibly execute immediately after other loops, to attribute that activity to a specific non-LIIS loop.

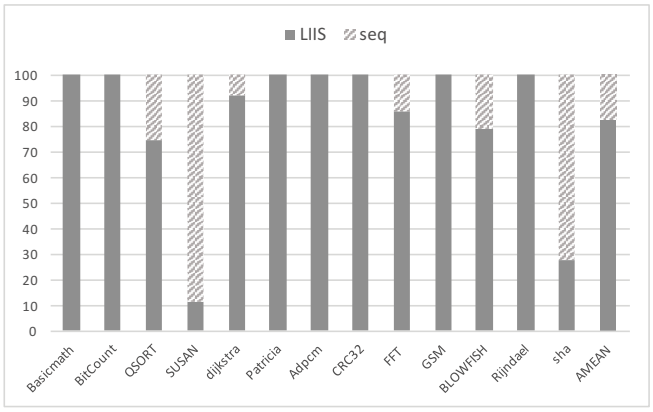

Fig. 5. Profiled time attributed through LIIS and Sequence mechanisms.

Fig. 5 shows how much of the profiled execution time is attributed through each of these mechanisms (LIIS and Sequence). On average, 82% of the execution time is attributed through LIIS, and some applications spend nearly all of their execution time in LIIS loops. However, in several applications (especially Susan and SHA) most of the execution time is attributed through the Sequence mechanism, and almost all of this attribution is correct (see Fig. 4). In general, we observed that Sequence-based attribution of profiled time is highly accurate, as long as the execution contains enough LIIS recognitions to constrain the set candidate loops to which the (non-LIIS) spectrum may possibly be attributed.

D. Accuracy for Loop Exit/Entry Time Profiling

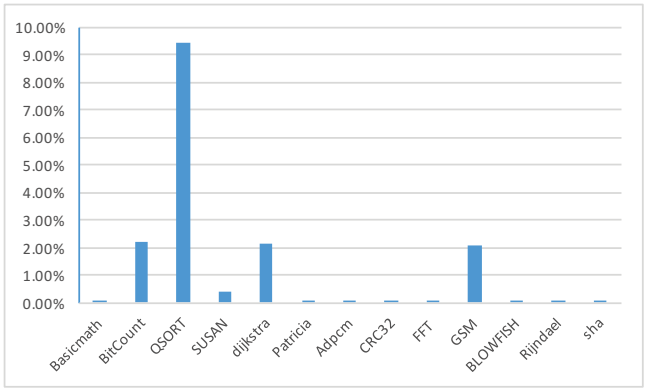

Fig. 6. Standard error for loop start/end times, normalized to loop duration.

In addition to overall accuracy with which execution time is attributed to specific loops, we measured how accurate our method is at determining the exact time when a loop is entered and exited. Specifically, if in the profiled run some loop "Loop1" starts at time t_1, and our Spectral Profiling implementation identifies t_1' as the start time, then the difference between t_1' and t_1 is the deviation (error) for this "sample" in this experiment. The samples in this experiment

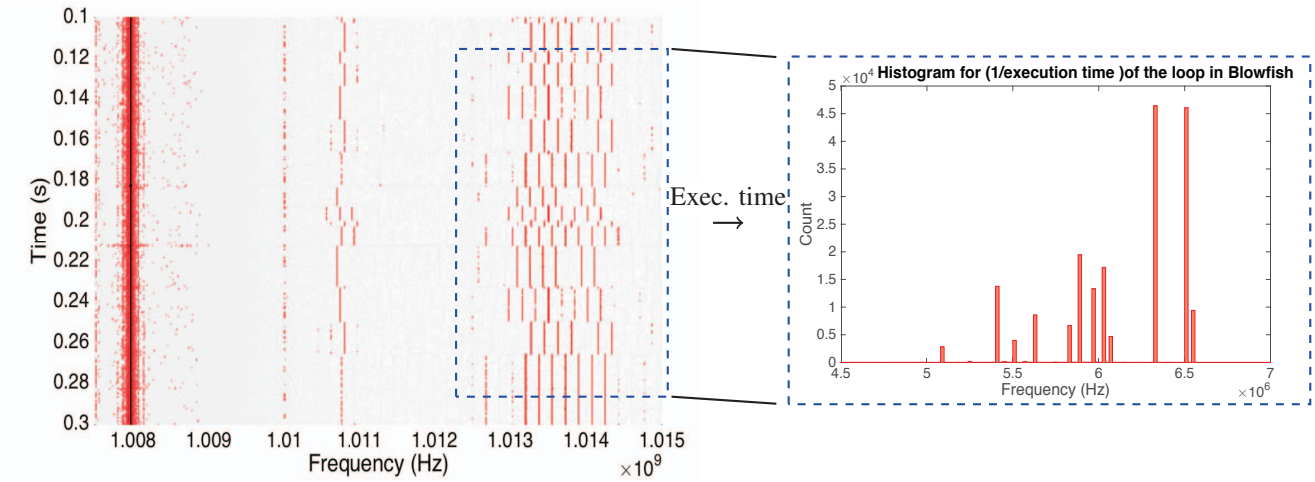

Fig. 7. Spectrogram and per-iteration execution time for a loop in *Blowfish*.

are loop start and end times for all dynamic instances of all loops executed in the application, and for each application we report the standard deviation across these samples, normalized to the average duration of the loop. Because we need the actual start/end times for each loop to compute the error, we only perform this measurement in simulation (where we can get the actual loop entry/exit times without changing the timing of the execution itself). The per-application results of this experiment are shown in Fig. 6, and across all benchmarks, the average of these normalized standard deviations is 1.42%.

E. Runtime Behavior of Loops

In addition to providing useful information about which regions of the code are hot and how much time is spent in each region, Spectral Profiling can also exploit the shape of the spikes in the spectrum to tell us the runtime behavior of the each loop. For example, sharp spikes indicate that almost all iterations of the loop take same amount of time. Conversely, having a wide spike, or a group of spikes, indicates that different iterations of the loop have different execution times. Such variation in per-iteration execution time could occur in outer loops when the number of iterations in their inner loops varies, and even in inner loops due to architectural events (e.g. cache misses, branch miss-predictions, etc) or differences in control flow among loop iterations. Identifying loops with unusually large per-iteration performance variation may help programmers identify performance problems, e.g. problems caused by unexpectedly large number of architectural events, unexpectedly frequent use of a long and seemingly unlikely program path within the loop body, etc.

To illustrate how Spectral Profiling can help understand performance of a loop, Fig. 7 shows the spectrogram (how the spectrum changes over time, where the spectrum is displayed horizontally and elapsed time is shown from bottom to top) for one loop in the "Blowfish" benchmark, for the real-system EM signal without any markers. We also show a histogram of actual per-iteration execution times in this loop, obtained by the markers during the execution with same inputs with markers. As seen in this figure, the duration and intensity of spikes in the frequency spectrum indicate how often (and when during the loop) different per-iteration execution times occur. In this loop, the variation in per-iteration execution time is caused by cache misses on one memory access instruction and branch mispredictions on two difficult-to-predict branch instruction in the loop body.

F. Effects of Changing Architecture

To show that ability to benefit from Spectral Profiling is not taking advantage of any specific architectural property of target machine, Fig. 8 shows the spectrogram for a same application, using the same inputs, on two different simulated systems:

Configuration	Clock	Core	Caches
A (Simple CPU)	50 MHz	InO 1-Wide	4 KB L1, No L2
B (Modern CPU)	1.8 GHz	OoO 4-Wide	32 KB L1, 64 MB L2

In this run, the application executes seven loops. The first loop is a nested loop that's why its signature is poorly defined at lower frequencies (which correspond to to the outer loop) with a sharp spike that corresponds to the inner loop at around 7.8 MHz in Config A and 81MHz in Config B. The remaining six loops each have a well-defined frequency, and can be clearly seen in spectrograms for both Config A and Config B. The vertical lines in the Config B spectrogram appear weaker because the spectral power of each spectral spike is distributed across a narrow frequency band as the out-of-order execution engine introduces slight variation among per-iteration execution times in each loop. However, spikes are still easily identifiable in both spectrograms, and allow us to attribute execution time to each of these seven loops, and also to determine their per-iteration execution time and its variation.

G. Size of the SFFT Window

Ideally, the profiled periodic activity lasts much longer than the window we use to compute the spectrum, so the "blurring" at the beginning and end of the activity introduces an error that

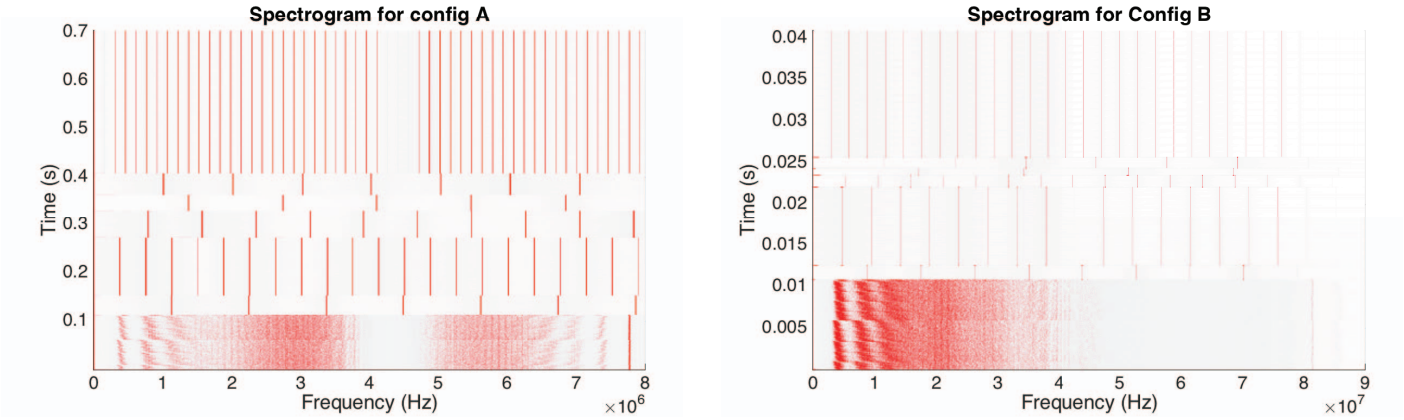

Fig. 8. Spectrograms from two different system configurations for *Bitcnt* benchmark for large size input.

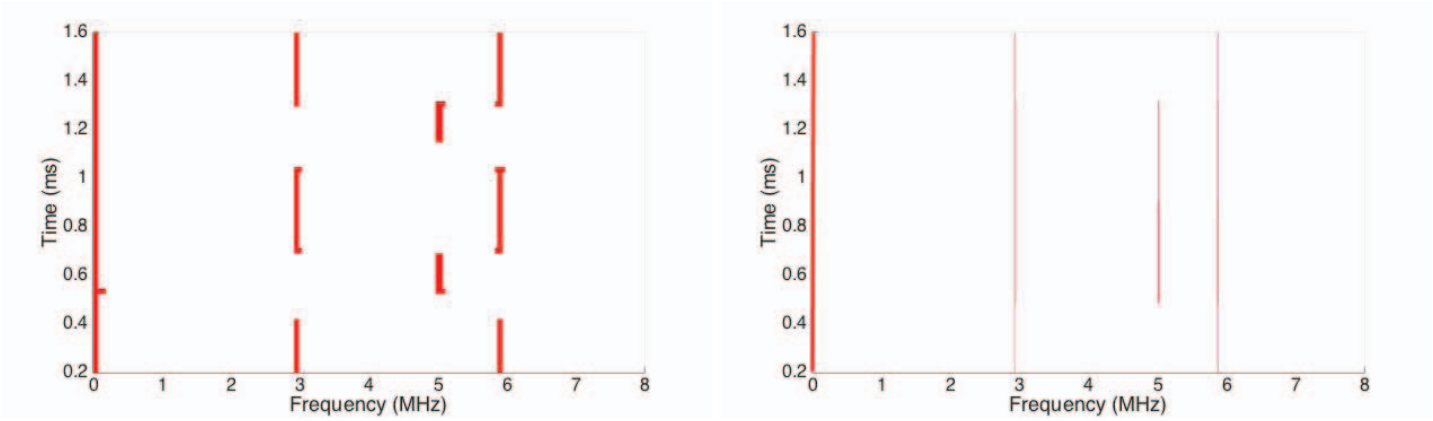

Fig. 9. Spectrograms for SFFT windows of 50us (left) and 500us (right), for the *Blowfish* benchmark with the large-size input.

is negligible relative to the duration of the activity. However, as described in Section III, windows that are too short provide low spectral resolution, i.e. they make it more difficult to tell different spectra apart. Consequently, the window size is a compromise between these two considerations. To illustrate this, Fig. 9 shows the spectrogram when the window is 500 μs and when it is 50 μs. The application spends most of its time in two loops, one with a frequency close to 3 MHz (the spectrogram also shows its harmonic that is close to 6 MHz), and the other loop has a frequency close to 5 MHz. Both spectrograms are derived from simulation-based power signals for the same simulation run. As can be seen in the figure, the shorter window allows us to clearly identify transitions between these loops. The longer window, however, sometimes (e.g. for 0.6 ms to 1.2 ms on the spectrogram) only indicates that both loops are active during the interval. However, note that the vertical lines in the short-window spectrogram are thicker – this indicates that the frequency bins are wider, i.e. there is less spectral resolution. In this particular example, the frequencies of the loops are far apart, so even the 50 μs window provides enough spectral resolution to distinguish them. This indicates that adaptive window size can improve Spectral Profiling accuracy beyond what is shown in this paper, where each experiment used a fixed window size.

V. Related Work

Information obtained through program profiling is useful for code optimization (e.g., [46]), testing and debugging (e.g., [47]), and software maintenance (e.g., [48]), and much effort has been invested in tools, algorithms, and even hardware support for profiling (e.g. [1], [2], [7], [8], [9], [10]). Profilers typically use instrumentation [49], [50], [1], [3], [5], [6] to count how often each part of the code is executed, or even to account for actual execution time, e.g. by reading a cycle-resolution counter [43]. Another approach is to periodically interrupt the program [2], [7], [8], [9], [13] and obtain samples of the program counter (PC) during execution, and use the distribution of these samples as an approximate distribution of where time was spent in the program. Hardware support [8], [11], [12], [13], [14] has been added to many processor specifically to reduce profiling overheads.

However, the overheads introduced by profiling can still be problematic. These overheads can be especially high if we need to profile the per-iteration behavior if tight (only a few instructions per iteration) loops. Even when overheads are a relatively small percentage of the execution time, they might fundamentally change the behavior of the application, especially in real-time and cyber-physical systems where the

application may switch to less demanding algorithms to avoid missing deadlines. In embedded and IoT systems, another problem is the lack (or limited) debugging infrastructure, resources (e.g. memory) to store profiling information, and the additional overheads and perturbation introduced by sending the profiling information out of the profiled system. Some of these problems can be addressed by adding Trace/Debug interfaces, but they often change the form factor of the device and limit its usability.

To our knowledge, the only other approach that attributes side-channel signals to specific parts of the program is ZOP [36], which uses time-domain correlation (i.e. correlation of signal samples obtained from the digital-to-analog converter) at the granularity of acyclic paths (usually one of a few basic blocks). As a result, ZOP requires orders of magnitude more computational effort and has only been evaluated for very brief runs. In contrast, Spectral Profiling exploits the connection between common program behaviors (loops) and the resulting EM signals to provide at-speed profiling of arbitrarily long program runs.

VI. Conclusions

This paper presents Spectral Profiling, a new method for profiling program execution without instrumenting or otherwise affecting the profiled system. Spectral Profiling monitors EM emanations unintentionally produced by the profiled system, looking for spectral "spikes" produced by periodic program activity (e.g. loops). This allows Spectral Profiling to determine which parts of the program have executed at what time and, by analyzing the frequency and shape of the spectral "spike", obtain additional information such as the per-iteration execution time of a loop. The key advantage of Spectral Profiling is that it can monitor a system as-is, without program instrumentation, system activity, etc. associated with the profiling itself, i.e. it completely eliminates the "Observer's Effect" and allows profiling of programs whose execution is performance-dependent and/or programs that run on even the simplest embedded systems that have no resources or support for profiling. We evaluate the effectiveness of Spectral Profiling by applying it to several benchmarks from MiBench suite on a real system, and also on a cycle-accurate simulator. Our experimental results show that our current implementation of Spectral Profiling on average correctly attributes 93% of execution time when applied to EM emanations from an actual IoT device, and we confirm the versatility of the approach by also successfully applying it to the power signal produced through cycle-accurate simulation of several different architectures, from sophisticated out-of-order cores to simple in-order cores. Additionally, our finding confirm that Spectral Profiling yields additional useful information about the runtime behavior of loops. Overall, Spectral Profiling can be used for profiling in systems where profiling infrastructure is not available, or where profiling overheads may perturb the results too much ("Observer's Effect").

Acknowledgment

This work has been supported in part by NSF grants 1563991, 1318934, and 1320717, AFOSR grant FA9550-14-1-0223, and DARPA LADS contract FA8650-16-C-7620. The views and findings in this paper are those of the authors and do not necessarily reflect the views of AFOSR, NSF, or DARPA.

References

[1] J. Demme and S. Sethumadhavan, "Rapid identification of architectural bottlenecks via precise event counting," in *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*, 2011.

[2] T. Moseley, D. A. Connors, D. Grunwald, and R. Peri, "Identifying potential parallelism via loop-centric profiling," in *Proceedings of the 4th International Conference on Computing Frontiers*, 2007.

[3] A. Ketterlin and P. Clauss, "Profiling data-dependence to assist parallelization: Framework, scope, and optimization," in *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2012.

[4] Y. Sato, Y. Inoguchi, and T. Nakamura, "On-the-fly detection of precise loop nests across procedures on a dynamic binary translation system," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, 2011.

[5] Q. Zhao, I. Cutcutache, and W. Wong, "Pipa: Pipelined profiling and analysis on multi-core systems," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2008.

[6] S. Wallace and K. Hazelwood, "Superpin: Parallelizing dynamic instrumentation for real-time performance," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2007.

[7] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S.-T. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous profiling: Where have all the cycles gone?," in *ACM Trans-actions on Computer Systems*, 1997.

[8] Intel-Corporation, "Intel vtune amplifier." https://software.intel.com/en-us/intel-vtune-amplifier-xe/details, accessed April 2, 2016.

[9] X. Yang, S. M. Blackburn, and K. S. McKinley, "Computer performance microscopy with shim," in *ACM/IEEE International Symposium on Computer Architecture (ISCA)*, 2015.

[10] S. L. Graham, P. B. Kessler, and M. K. McKusiclc, "Gprof: A call graph execution profiler," in *Proceedings of the SIGPLAN Symposium on Compiler Construction*, 1982.

[11] Intel-Corporation, "Precise-event-based-sampling on intel processors." http://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf, accessed April 2, 2016.

[12] Intel-Corporation, "Intel performance counter monitor." https://software.intel.com/en-us/articles/intel-performance-counter-monitor, accessed April 2, 2016.

[13] Linux, "Linux kernel profiling with perf." https://perf.wiki.kernel.org/index.php/Main_Page, accessed April 3, 2016.

[14] ARM, "Arm performance monitor unit." http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388f/Bcgddibf.html, accessed April 2, 2016.

[15] H. Highland, "Electromagnetic radiation revisited," *Computers and Security*, pp. 85–93, dec 1986.

[16] D. Agrawal, B. Archambeault, J. R. Rao, and P. Rohatgi, "The EM side-channel(s)," in *Proceedings of Cryptographic Hardware and Embedded Systems (CHES)*, pp. 29–45, 2002.

[17] A. Zajić and M. Prvulovic, "Experimental demonstration of electromagnetic information leakage from modern processor-memory systems," *IEEE Transactions on Electromagnetic Compatibility*, vol. 56, no. 4, pp. 885–893, 2014.

[18] K. Gandolfi, C. Mourtel, and F. Olivier, "Electromagnetic analysis: concrete results," in *Proceedings of Cryptographic Hardware and Embedded Systems - CHES 2001*, pp. 251–261, 2001.

[19] M. G. Khun, "Compromising emanations: eavesdropping risks of computer displays," *The complete unofficial TEMPEST web page: http://www.eskimo.com/˜joelm/tempest.html*, 2003.

[20] H. Sekiguchi and S. Seto, "Measurement of radiated computer rgb signals," *Progress in Electromagnetic Research C*, pp. 1–12, 2009.

[21] Y. Suzuki and Y. Akiyama, "Jamming technique to prevent information leakage caused by unintentional emissions of pc video signals," in *Electromagnetic Compatibility (EMC), 2010 IEEE International Symposium on*, pp. 132–137, 2010.
[22] M. G. Kuhn, "Compromising emanations of lcd tv sets," *IEEE Transactions on Electromagnetic Compatibility*, pp. 564–570, 2013.
[23] T. Plos, M. Hutter, and C. Herbst, "Enhancing side-channel analysis with low-cost shielding techniques," in *Proceedings of Austrochip*, 2008.
[24] F. Poucheret, L. Barthe, P. Benoit, L. Torres, P. Maurine, and M. Robert, "Spatial EM jamming: A countermeasure against EM Analysis?," in *Proceedings of the 18th IEEE/IFIP VLSI System on Chip Conference (VLSI-SoC)*, pp. 105–110, 2010.
[25] H. Tanaka, "Information leakage via electromagnetic emanations and evaluation of Tempest countermeasures," in *Lecture notes in computer science, Springer*, pp. 167–179, 2007.
[26] Y. ichi Hayashi, N. Homma, T. Mizuki, H. Shimada, T. Aoki, H. Sone, L. Sauvage, and J.-L. Danger, "Efficient evaluation of em radiation associated with information leakage from cryptographic devices," *IEEE Transactions on Electromagnetic Compatibility*, vol. 55, no. 3, pp. 555–563, 2013.
[27] H. Sekiguchi and S. Seto, "Study on maximum receivable distance for radiated emission of information technology equipment causing information leakage," *IEEE Transactions on Electromagnetic Compatibility*, vol. 55, no. 3, pp. 547–554, 2013.
[28] Y. ichi Hayashi, N. Homma, T. Mizuki, T. Aoki, H. Sone, L. Sauvage, and J.-L. Danger, "Analysis of electromagnetic information leakage from cryptographic devices with different physical structures," *IEEE Transactions on Electromagnetic Compatibility*, vol. 55, no. 3, pp. 571–580, 2013.
[29] S. Clark, H. Mustafa, B. Ransford, J. Sorber, K. Fu, and W. Xu, "Current events: Identifying webpages by tapping the electrical outlet," in *Computer Security- ESORICS 2013* (J. Crampton, S. Jajodia, and K. Mayes, eds.), vol. 8134 of *Lecture Notes in Computer Science*, pp. 700–717, Springer Berlin Heidelberg, 2013.
[30] C. R. A. Gonzlez and J. H. Reed, "Power fingerprinting in sdr integrity assessment for security and regulatory compliance," *Analog Integrated Circuits and Signal Processing*, vol. 69, no. 2-3, pp. 307–327, 2011.
[31] S. S. Clark, B. Ransford, A. Rahmati, S. Guineau, J. Sorber, W. Xu, and K. Fu, "Wattsupdoc: Power side channels to nonintrusively discover untargeted malware on embedded medical devices," in *USENIX Workshop on Health Information Technologies*, 2013.
[32] R. Callan, A. Zajić, and M. Prvulovic, "A practical methodology for measuring the side-channel signal available to the attacker for instruction-level events," in *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.
[33] R. Callan, N. Basta, A. Zajic, and M. Prvulovic, "A new approach for measuring electromagnetic side-channel energy available to the attacker in modern processor-memory systems," in *Proceedings of the 9th European Conference on Antennas and Propagation (EuCAP)*, 2015.
[34] R. Callan, A. Zajic, and M. Prvulovic, "FASE: Finding Amplitude-modulated Side-channel Emanations," in *the 42nd International Symposium on Computer Architecture (ISCA)*, 2015.
[35] R. Callan, N. Popovic, A. Daruna, E. Pollmann, A. Zajic, and M. Prvulovic, "Comparison of electromagnetic side-channel energy available to the attacker from different computer systems," in *Proceedings of the 2015 IEEE International Symposium on Electromagnetic Compatibility (EMC)*, 2015.
[36] R. Callan, F. Behrang, A. Zajic, M. Prvulovic, and A. Orso, "Zero-overhead profiling via em emanations," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, 2016.
[37] M. R. Guthaus, J. S. Pingenberg, D. Emst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the IEEE International Workshop on Workload Characterization*, 2001.
[38] E. Sejdić, I. Djurović, and J. Jiang, "Time–frequency feature representation using energy concentration: An overview of recent advances," *Digit. Signal Process.*, vol. 19, pp. 153–183, Jan. 2009.
[39] Olimex, "A13-olinuxino-micro user manual." https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino-MICRO/open-source-hardware, accessed April 3, 2016.
[40] ARM, "Arm cortex a8 processor manual." https://www.arm.com/products/processors/cortex-a/cortex-a8.php, accessed April 3, 2016.
[41] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Prvulovic, L. Ceze, S. Sarangi, P. Sack, K. Strauss, and P. Montesinos, "SESC simulator," January 2005. http://sesc.sourceforge.net.
[42] AARONIA, "Datasheet: Rf near field probe set dc to 9ghz." http://www.aaronia.com/Datasheets/Antennas/RF-Near-Field-Probe-Set.pdf, accessed April 6, 2016.
[43] G. Paoloni, "White paper: How to benchmark code execution times on intel ia-32 and ia-64 instruction set architectures," tech. rep., Intel Corporation, September 2010.
[44] G. Reinman and N. Jouppi, "Cacti 2.0: An integrated cache timing and power model," *Technical Report*, 2000.
[45] D. Brooks, V. Tiwari, and M. Martonosi in *ACM/IEEE International Symposium on Computer Architecture*, ISCA-27, pp. 83–94, 2000.
[46] S. Debray and W. Evans, "Profile-guided code compression," in *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*, 2002.
[47] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani, "HOLMES: Effective Statistical Debugging via Efficient Path Profiling," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, 2009.
[48] M. D. Ernst, J. CockrelI, W. G. Griswold, , and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," in *Proceedings of the 21st International Conference on Software Engineering (ICSE)*, 1999.
[49] T. Ball and J. R. Larus, "Efficient Path Profiling," in *Proceedings of the 29th annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 1996.
[50] R. Joshi, M. D. Bond, and C. Zilles, "Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, 2004.