Performance Data Collection Using a Hybrid Approach

Edu Metz
Nokia Research Center
5 Wayside Road,
Burlington, MA 01803, USA
Edu.Metz@nokia.com

Raimondas Lencevicius
Nokia Research Center
5 Wayside Road,
Burlington, MA 01803, USA
Raimondas.Lencevicius@nokia.com

Teofilo F. Gonzalez
Department of Computer Science
University of California
Santa Barbara, CA 93106, USA
teo@cs.ucsb.edu

ABSTRACT

Performance profiling consists of monitoring a software system during execution and then analyzing the obtained data. There are two ways to collect profiling data: event tracing through code instrumentation and statistical sampling. These two approaches have different advantages and drawbacks. This paper proposes a hybrid approach to data collection that combines the completeness of event tracing with the low cost of statistical sampling. We propose to maximize the weighted amount of information obtained during data collection, show that such maximization can be performed in linear time or is NP-hard depending on the data collected and the collection implementation. We propose an approximation algorithm for NP-hard case. Our paper also presents an application of the formal approach to an example use case.

Categories and Subject Descriptors

D.2.5 Testing and Debugging

General Terms

Algorithms, Measurement, Performance.

Keywords

Profiling, sampling, tracing.

1. INTRODUCTION

As the complexity of embedded software systems grows, performance profiling is becoming increasingly more important. Performance profiling of embedded software systems requires data collection with low overhead and high information completeness.

Performance profiling consists of monitoring a software system during execution and then analyzing the obtained data. There are two ways to collect profiling data: event tracing through code instrumentation and statistical sampling. Event tracing is generally more intrusive to software system execution, but allows the profiler to record all events of interest. Statistical sampling may be less intrusive, but cannot provide complete execution information.

This paper proposes and explores a hybrid approach to data collection that combines the completeness of event tracing with the low cost of statistical sampling.

Section 2 describes and compares typical data collection methods for profiling: event tracing and statistical sampling. Section 3 presents the new hybrid approach and motivating examples of its use. Section 4 proposes how to maximize the information amount by selecting which events to trace and which to sample. Section 5 discusses a real-world example of hybrid profiling. The paper concludes with related work, future work and conclusions.

2. PERFORMANCE DATA COLLECTION

Performance profiling determines where a software system spends its execution time [6]. Performance profiling requires data to be collected during program execution. Such data collection can be done either by event tracing or by statistical sampling. The following subsections briefly describe and compare two methods.

2.1. Event tracing

Event tracing records events that occur during system execution. Event tracing can track various events, such as task switches, component entries and exits, function calls, branches, software execution states, message communication, input/output, and resource usage.

Tracing using software techniques[1] requires changes to the software system usually called *instrumentation*. Instrumentation can be inserted into various program representations: source code, object code, byte code, and executable code. It can be inserted before or during program execution. Trace instrumentation can be added manually, semi-automatically or automatically. Automatization of the instrumentation may be complex.

[1] Although tracing and sampling using hardware monitors are possible, they are not discussed in this paper, since hardware monitors usually have no overhead to the profiled system.

2.2. Statistical sampling

Statistical sampling relies on intermittent access to the software system to record its current state. Sampling can record various information: program counter (execution location), function call stack, scheduled or blocked tasks, active peripherals and so on. Sampling can be done strictly periodically or with certain randomness. In periodic real-time systems, the sampling interval needs to be randomized to avoid sampling the same periodic software entity at every sampling point.

The simplest forms of sampling do not require any software modifications. A sampler simply copies the content of certain processor registers to memory. In more complex sampling, the software system may need to be interrupted to record the needed information.

2.3. Comparison of tracing and sampling

In event tracing every occurrence of an event creates a record. So event tracing is characterized by the completeness of knowledge: if an event was recorded, it did occur; if it was not recorded, it did not occur. Performance engineers can also learn exactly when each event occurred since every record is time stamped. This allows a complete analysis of event relationships in time, for example, the measurement of precise time distance between any two events. A performance engineer with a sufficiently detailed event trace – ideally, processor instruction trace, can reconstruct the dynamic behavior of a software system.

Sampling yields only a statistical measure of the software's execution patterns. It does not provide completely precise numbers: if an event does not occur in a sampling log, there is no guarantee that it did not occur in execution. Therefore sampling may not be useful for situations that need to track exact numbers of events, for example, a singleton message to a task or an exact relationship between requests and acknowledgements.

Sampling is not a good approach when event causality is analyzed. Although it may extract a function call stack at the sample time, it cannot track all function calls or message exchanges. A performance engineer who needs a complete message sequence chart or component interaction graph might be better off choosing event tracing.

Software tracing requires users to spend time instrumenting the software system. A performance engineer would usually spend significantly less time to achieve sampling than to instrument the software system for tracing.

Both event tracing and sampling may affect the performance of the software system, thereby distorting its execution [14]. Not only do they add overhead, they can also change the behavior of the software system because of additional memory accesses and input/output [12]. In real-time software systems, the overhead can cause real-time constraint violations. Therefore, it is important to limit the intrusion by minimizing the overhead [2][9].

In tracing one way to achieve this is by reducing the number of events traced. However, performance engineers have to choose carefully, since omitting events from tracing also reduces the amount of information available. For example, if only "on" and "off" events are traced in a peripheral, it is no longer possible to detect and map the peripheral's different "on" modes to differences in the system's power consumption. In choosing the instrumentation granularity it is important to address the trade-off between the amount of event information required and the performance impact of the trace instrumentation. This may be hard even for an experienced performance engineer.

On the other hand, the overhead of sampling may be orders of magnitude below the overhead of tracing. For example, branch tracing may require overhead of 10 times the original execution, function tracing may require overheads up to 2 times, while sampling at up to one thousand samples a second may have an overhead of less than 1% [1]. (This estimation assumes a 100Mhz processor and 1000 cycles of work per sample, which is enough to read the address of the currently executed instruction and save this information. Using symbol information generated at compile time, the profiler can later correlate the recorded sample with the source code.) Sampling the execution of software in a mobile device executing real-time tasks may be the only way to obtain information about long-running functions without causing the software to miss real-time deadlines due to tracing overhead.

The data volume associated with event tracing can be very large [14]: up to a gigabyte per second traced if we consider executed instruction traces and tens of megabytes per second even in less frequent traces. This can cause a problem in devices that do not have large and fast storage or external network interfaces. Sampling may be done at a lower frequency and produce much less data than event tracing—a positive in storage-limited devices.

However, the sampling frequency determines the granularity of the gathered information. In addition, the duration for which the software system executes directly relates to the number of samples collected. A sampling profiler requires software systems to execute over a reasonable period of time to ensure accuracy [13]. The goals of a performance engineer may require high sampling frequency that negates the low overhead and small data production of sampling.

For small routines, event tracing may not yield an accurate time comparison with larger routines. A small routine may suffer much higher relative overhead from tracing than a larger routine. If this is ignored, a great deal of effort may be wasted optimizing routines that are not real performance bottlenecks.

Sampling may not be able to detect frequently executed routines whose execution times are smaller than the sampling frequency. In addition, manual trace instrumentation usually tracks application-specific events that could be difficult to capture by sampling. For example, detecting a transition from a single-person voice call to a conference call may require event tracing.

3. HYBRID DATA COLLECTION

Let us summarize the previous section. Event tracing yields the most detailed and complete system execution data. However, it takes time to instrument software, tracing has a high overhead and may change the behavior of the software system [12]. Statistical sampling is simple to use and less intrusive to software system execution, but does not provide causality relationships and exact data.

Embedded software systems, such as mobile devices, have real-time constraints and therefore require performance profiling methods with low overheads. On the other hand, performance analysis of such devices often involves causality relationships and precision requirements. For example, a

performance engineer needs to know exactly when a multiplayer-game task starts processing a message that changes the game environment, since this may point to the cause of performance bottleneck evidenced by numerous file accesses.

Often neither event tracing nor statistical sampling can satisfy such conflicting requirements. The problem is further compounded by the fact that test runs are not entirely deterministic in mobile devices due to interactions with other systems such as mobile network elements. Therefore, performance data cannot be collected during multiple test runs, but instead needs to be collected during a single test run.

To collect performance data of embedded software systems with low overhead and adequate completeness, we propose to use a middleweight approach which is a hybrid of heavyweight event tracing and lightweight statistical sampling. Only a subset of all events is traced, providing limited completeness and causality information. Additional information is obtained through sampling.

To apply our method, a performance engineer has to determine which part of the performance data should be collected using event tracing and which using statistical sampling. The following subsections provide couple examples of hybrid profiling and decisions of what to trace and what to sample. Section 4 introduces a formal approach to making tracing and sampling split decisions, so that the information amount in collected data is maximized.

3.1. Processor time profiling

When the goal of a performance engineer is to determine which software components and subsystems spend the most time running on a processor, statistical sampling can provide most of the needed information. It can reveal the approximate amount of time spent in a component, such as a task, module or function. Event tracing can then be used to supplement this information in a couple of areas. First, it can precisely identify switches of very high level components, such as tasks. Second, it can demonstrate the component execution causality by tracking message exchanges. For example, consider the synchronization between tasks A and B in Figure 1.

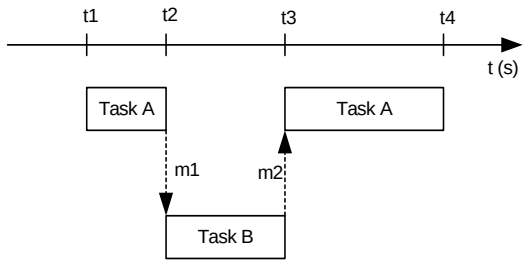

Figure 1: Task state synchronization

After sending message m1, task A enters a wait state where it waits for a state synchronization callback m2 from task B before continuing its execution. Here, event tracing can record and timestamp the sending of messages m1 and m2, while sampling can provide more in depth performance data during time intervals [t1, t2], [t2, t3], [t3, t4]. In this case, sampling on its own would not be enough to provide the crucial synchronization information.

3.2. Resource usage and energy profiling

In mobile devices power consumption varies depending on the peripherals used [7]. During the system execution, software accesses peripherals. These accesses need to be recorded to determine when a peripheral is used. In resource usage and energy profiling, complete information about active and inactive peripherals is required. Event tracing needs to be used to track state transitions of Bluetooth, GPS or infrared subsystems. The intrusion cost of recording "on" and "off" events of peripherals is low since they occur infrequently.

Statistical sampling can complement event tracing by providing information that is too expensive to obtain using event tracing alone. For example, the processor power management puts the processor in a low power sleep mode when no software is scheduled to run. Unlike Bluetooth mode changes, the processor's transition to the sleep state may be too frequent and too expensive to track via instrumentation. Statistical sampling can reveal the processor's idle state with enough accuracy as long as the context switch time is an order of magnitude larger than the sampling frequency.

Another opportunity for sampling is presented by devices with multiple active modes. The overhead of tracing every state transition of a peripheral may be too high. While tracing could provide information about major "on" and "off" states, sampling could complement this information with infrequent samples of secondary states allowing more precise system mapping than achieved with tracing alone.

4. MAXIMIZING INFORMATION IN DATA COLLECTION

As shown in the examples above, to use the hybrid approach, performance engineer needs to decide which events to trace and which to sample. In other words, hybrid data collection is based on splitting all the data into two parts: the part collected using tracing and the part collected using sampling. We propose to perform the split in such a way that the profiling overhead is limited and the amount of information collected is maximized. The remainder of this section formalizes this approach.

4.1. Base case

This subsection describes the base case of the problem and its solution.

Let us define Overhead as the slowdown of the program due to the data collection. It is defined as a ratio:

$$\text{Overhead} = \frac{\text{execution time with data collection}}{\text{original execution time}} = 1 + \frac{\text{instrumentation or sampling time}}{\text{original execution time}}$$

Assume a model where a program can tolerate a uniform overhead that is not larger than MaxAllowedOverhead.

If the overhead of collecting all data using tracing is less or equal to MaxAllowedOverhead then tracing alone can be used. No sampling is needed and our problem of splitting data collection into sampling and tracing is solved.

However, if the overhead of collecting all data using tracing is greater than MaxAllowedOverhead, some data needs

to be sampled at a lower frequency than tracing to lower the overhead to MaxAllowedOverhead.

Assume that we have n data classes of events $C_1 \ldots C_n$. Events in each of these classes occur at a frequency of $F_1 \ldots F_n$ (frequencies are measured in Hz = 1/sec). Assume that reporting any event, whether it is traced event or sampled event takes the same amount of time T_{report} (measured in seconds).

Using the notation introduced, we can rewrite Overhead definition as follows:

$$Overhead = 1 + \frac{\text{instrumentation or sampling time}}{\text{original execution time}} =$$

$$1 + \frac{(\sum_{i=1}^{n} F_i) * T_{report} * (\text{original execution time})}{\text{original execution time}} =$$

$$1 + (\sum_{i=1}^{n} F_i) * T_{report}$$

As mentioned above, our goal is to reduce the overhead to MaxAllowedOverhead. The only control we have is over the frequencies $F_1 \ldots F_n$. These frequencies can be reduced by sampling at lower frequencies, thereby reducing the overhead. Let us call the reduced frequencies $F_{red1} \ldots F_{redn}$. Using reduced frequencies, the overhead is $1 + (\sum_{i=1}^{n} F_{redi}) * T_{report}$

We can reduce the overhead by sampling; however, sampling at a lower rate than tracing loses some information. How do we decide which event classes to sample and which to trace? How do we decide how much the sampling frequency should be reduced for each event class? For this we introduce a metric that measures the amount of information available in the collected data.

Assume that each event class $C_1 \ldots C_n$ has information weight $W_1 \ldots W_n$. Information weight expresses the importance of gathering larger percentage of events of a class. Larger relative weight means that the event class carries more information and consequently is more important to the user. Users assign information weights for different event classes. For example, if file-read events were very important, the users would select a high information weight to the file-read event class.

Information weights allow us to introduce a metric to measure the amount of information available in the collected data. We call it *information value* and define it as follows:

InformationValue = $F_{red1} * W_1 + \ldots + F_{redn} * W_n$

Intuitively, the information value is a weighted sum of the frequencies of different event classes. Users want to maximize this value. This value can be increased by increasing the frequencies $F_{red1} \ldots F_{redn}$. However, the overhead constraint does not allow unlimited increase of the frequencies. This leads to the following problem:

Problem 1.

Maximize InformationValue $= \sum_{i=1}^{n} F_{red\,i} W_i$ so that

$$Overhead = 1 + (\sum_{i=1}^{n} F_{redi}) * T_{report} \le MaxAllowedOverhead$$

To maximize the information value, frequencies should be increased as much as possible, so Overhead increases until it is equal to MaxAllowedOverhead. By moving all constants of the constraint to the right hand side, we get the following equation:

$$\sum_{i=1}^{n} F_{red\,i} = \frac{\text{MaxAllowedOverhead} - 1}{T_{report}},$$

$$and \;\; F_{red\,i} \le F_i, \forall i = 1..n$$

To simplify the equation, let us define MaxF as the right hand side constant (MaxAllowedOverhead - 1)/T_{report}. MaxF is known before making the sampling/tracing decision. Intuitively it represents the maximum frequency of information retrieval that does not exceed the maximum allowed overhead.

Problem 1 is a linear programming problem that can be solved using any available linear programming solver [3][8]. The solver produces a set of concrete reduced frequencies $F_{red1} \ldots F_{redn}$ that maximize the information value of collected data while observing the overhead constraints. This is exactly what was needed.

A more efficient way to solve this problem is by viewing the problem as a version of the continuous Knapsack problem. To see this we replace in Problem 1 F_{redi} by $F_i\, x_i$, where $0 \le x_i \le 1$ for $1 \le i \le n$. The problem becomes

Maximize InformationValue $= \sum_{i=1}^{n} W_i\, F_i\, x_i$ so that

$$\sum_{i=1}^{n} F_i x_i \le MaxF \;\; \text{and } 0 \le x_i \le 1.$$

The continuous Knapsack problem is defined as "maximize $\sum_{i=1}^{n} P_i\, x_i$ so that $\sum_{i=1}^{n} S_i\, x_i \le B$, and $0 \le x_i \le 1$, where all the profits P_i are positive, the size of the objects S_i are positive and the capacity of the knapsack is at most B". As pointed in [11] an optimal solution can be obtained by the greedy strategy that "considers objects in nonincreasing order of profit density P_i / S_i; if there is enough remaining capacity to accommodate the object, put it in; if not, put a fraction to fill the knapsack."

In our problem the objects are considered in nonincreasing order of $\frac{P_i}{S_i} = \frac{F_i\, W_i}{F_i} = W_i$. The above greedy strategy can be implemented to run in O(n log n) time. By using the O(n) time median finding algorithm (selection algorithm in [11]) one may reduce the overall time complexity of the procedure to O(n) time.

As we mentioned before, the base case is applicable only if specific assumptions are satisfied. In the next subsection, we describe its applicability domain and build a different solution for another common domain.

4.2. Hybrid data collection using probing

Section 4.1 proposed a solution to the question which event classes to trace and which to sample. However, this solution is based on two assumptions that determine its applicability domain:

- It assumes that it is possible to sample events at any reduced frequency
- It assumes that sampling only samples a single data class

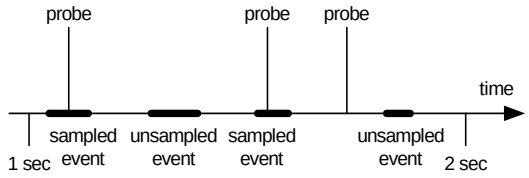

Figure 2: Event sampling

Consider Figure 2. It shows 1 second time interval with events marked as bold lines. There are four events on the figure and two of them are sampled. The event class frequency is F_i = 4 Hz and reduced frequency is F_{redi} = 2 Hz. However, consider the following question about the implementation of this scheme: how did the system sample these two events? To select two out of four events, the system had two options. First option: to detect all four events and only report two of them. This option satisfies our two assumptions and falls into the applicability domain of the section 4.1 solution. The second option: the system just checked for events at some frequency. If it detected an event, it reported it. If it did not detect an event, it did not report anything (see labels "probe" on the figure). This option, however, does not satisfy either of the two assumptions:

- If second option is used, the real sampling frequency on the figure is 3 Hz, not 2 Hz. There are 3 samples: two "positive" probes and one "negative" probe. All of them contribute to overhead. If we reduced the sampling frequency to 2 Hz, however, there is no guarantee that both probes would hit an event. If one of them does not hit an event, the information value cannot be calculated using 2 Hz frequency value, since only one event is reported.
- The probes that check the system for events of one class can check it for events of other classes too. In other words, the same probe can report events of multiple classes.

Since this approach of periodic probing to collect samples of data is pretty common, we need to extend our approach to cover this domain. First, let us introduce the following notions. Assume there is a single probe that probes and samples all events. The frequency of this probe is $F_{probing}$ (3 Hz in our example). It can be changed to change the overhead. Since there is no reduced sampling for each event class, we can only make a binary decision: to trace the event class or to sample it via probing. If we trace the event class, it contributes to the overhead with a frequency F_i. If we sample the event class via probing, it does not contribute to the per-class part of the overhead, since this part only includes tracing overhead now. The overhead of probing is independent of event classes and is common to all of them. The new expression for overhead is:

$$Overhead = 1 + (F_{probing} + \sum_{i=1}^{n} F_{istraced\,i}) * T_{report},$$

$$F_{istraced\,i} = F_i, \text{if } C_i \text{is traced},$$

$$F_{istraced\,i} = 0, \text{if } C_i \text{is not traced}, \forall i = 1..n$$

The information value function changes as well:

$$InformationValue = \sum_{i=1}^{n} F_{hits\,i} * W_i,$$

$$F_{hits\,i} = F_i, \text{if } C_i \text{is traced},$$

$$F_{hits\,i} = F_{probing} \times \frac{EventTime_i}{EventPeriod_i}, \text{if } C_i \text{is not traced},$$

$$\forall i = 1..n$$

$F_{hits\ i}$ gives the frequency of probe hitting class i events. Only the events hit by the probe or traced provide the information, so only these events are included into the calculation of the information value. Unsampled events (Figure 2) are excluded. In the formula for F_{hitsi} calculation, $EventTime_i$ is the average length of time for the event of data class i measured in seconds. $EventPeriod_i$ is the period of events in data class i. Their ratio gives a probability that the probe will hit the event of class i. By definition $EventPeriod_i = 1 / F_i$.

With these definitions we specify the new optimization problem.

Problem 2.

Maximize the information value with the constraint $Overhead \le MaxAllowedOverhead$.

Using MaxF defined earlier, we can rewrite the overhead constraint as:

Constraint 1.

$$F_{probing} + \sum_{i=1}^{n} F_{istraced\,i} = MaxF \qquad (1)$$

Problem 2 is solvable in $O(2^n)$ time by checking all possible decisions of whether each event class is traced or not. Each such decision determines corresponding $F_{istraced\ i}$. $F_{probing}$ can then be expressed and determined from constraint 1. Finally, $F_{hits\ i}$ and information value can be calculated. Is it possible to solve this problem more efficiently?

Theorem 1. Problem 2 is NP-hard.

Proof. Consider the situation where $EventTime_i/EventPeriod_i = 0$ for all i. We remove this restriction later. Let us set all W_i's to 1. In this case an optimal solution is one that selects a subset of the objects whose F_i's sum is as close to MaxF as possible. We reduce PARTITION [4] to this problem.

Partition is given objects $a_1 \ldots a_n$ with sizes $s(a_1) \ldots s(a_n)$. The goal is to partition $a_1 \ldots a_n$ into two subsets A_1 and A_2 such that the sum of the sizes of all the objects in A_1 equals exactly the sum of the sizes of all the objects in A_2.

The reduction sets F_i to $s(a_i)$ and MaxF is set to $(\sum_{i=1}^{n} s(a_i))/2$. If there is a partition, then an optimal solution

has value $(\sum_{i=1}^{n} s(a_i))/2$. Otherwise the optimal solution has a smaller value.

Since in our case $EventTime_i/EventPeriod_i$ cannot be exactly zero, we have to handle the case where all these ratios are positive. In this case we make the ratios very small. We can set all of them to $EventTime_i/EventPeriod_i < 1/MaxF$. By doing this we guarantee that the contribution from all the values of i such that $F_{istraced\ i}$'s = 0 will be less than one. One can show then that the above reduction applies to this case too. *QED*.

This theorem shows that the new problem is hard to solve when the number of event classes is large. However, we present a simple O(n log n) approximation algorithm that can find a solution with the objective function value within 50% of the objective function value of an optimal solution.

Assume without loss of generality that $W_1 \geq W_2 \geq \ldots \geq W_n$. Our approximation algorithm is given below.

Approximation Algorithm

for i = 0 to n do
  Let $S_{i,0}$ be the solution where C_j is not traced for all j,
    and if $i \neq 0$, then C_i is traced.
  Let $s_{i,0}$ be the objective function value of $S_{i,0}$.
end for
for j = 1 to n do
  let $S_{0,j} \leftarrow S_{0,j-1}$
  let $s_{0,j}$ be the objective function value of $S_{0,j}$
  if one can trace C_j in solution $S_{0,j}$ without making
      $F_{probing}$ negative then
      let C_j be traced in solution $S_{0,j}$
      let $s_{0,j}$ be the objective function value of $S_{0,j}$
end for
Output the best of the solutions $S_{i,0}$ for $1 \leq i \leq n$
 and $S_{0,j}$ for $1 \leq j \leq n$
end of algorithm

Theorem 2: The approximation algorithm generates a solution with objective function $\hat{f} > \frac{1}{2} f^*$, where f^* is the objective function value of an optimal solution.

Proof. The proof is presented in Appendix A.

By saving only the best solution so far generated and avoiding the copy of the previous solution to the new one, one can implement the approximation algorithm to take O(n log n) time.

4.3. Additional considerations

There are a number of details and considerations important to the application of the hybrid profiling. While sections 4.1 and 4.2 presented the main ideas of our approach, this section discusses details and special cases important in applying the approach.

4.3.1 Data non-deductability assumption

The approach proposed assumes that all the data to be gathered cannot be deduced from other data. For example, if function A always calls function B, we do not need to gather function B call data, since we know that it occurs whenever function A is executed. (There is some complexity here: we still need to gather function B call data if function B is called by other functions. Also we may need to gather function B call *time*, since this cannot be precisely determined from the function A call data). So we assume that the data we collect is *causally independent*, i.e. it cannot be determined from other collected data.

4.3.2 Simplification in event frequency model

Our event frequency model is simplified, because the events could be interrupted by other events. For example, processes in software system can interrupt other processes. If process activity is an event in process data class, the interruptions of other processes increase the number of distinct process data events. Interruptions by other events should be accounted for when specifying $EventTime_i$ and $EventPeriod_i$ values. Also such interruptions increase the cost of tracing, because the number of events increases, but they do not increase the cost of sampling. One way to deal with such interruptions is instead of actual period and execution times, take the "interrupted" periods and execution times. These may not have very regular periodicity, but may better represent event numbers.

4.3.3 Different cost sampling and tracing

There are situations when the cost of taking a single sample is different from the cost of taking a single trace event. In this case, instead of a single value T_{report} we have to consider sampling time $T_{sampling}$ and tracing time $T_{tracing}$. The information value function does not change, while the constraint for problem 2 becomes:

$$F_{probing} * T_{sampling} + \sum_{i=1}^{n} F_{istraced\ i} * T_{tracing} = MaxSlowdown - 1 \qquad (2)$$

The complexity of problem 2 with constraint (2) remains the same as earlier. Problem is still solvable in $O(2^n)$ time. Setting $T_{sampling}$ and $T_{tracing}$ to the same value allows for the same reduction to PARTITION in proof, showing that the problem remains NP-hard.

4.3.4 Negligible cost for detecting all events assumption

We mentioned that in the base case the sampling process somehow samples events at any reduced frequency. One way to do it is detecting all events of a class and only reporting some percentage of them. However, this assumes that detecting all events is negligibly cheap. Otherwise the overhead would not be proportional to the reduced frequency. It still would be dependent on original event frequency. For example, if we detected all four events in Figure 2 and reported only two, the cost should be proportional to the 2 Hz frequency of the reported events for the base case to apply. We can make this assumption in some cases. In a lot of modern implementations,

event detecting may be accomplished by simply inserting one or two lines of code. There are situations where reporting is much more costly than detecting, because data has to be immediately written into a file or to an outside system through a network. For example, when monitoring events on a mobile device, the memory space can be limited, and the data has to be written into a flash drive or to an external system through USB, Ethernet, BlueTooth, WLAN or cellular data connection. For example, in a mobile device authors considered, the reporting was over 30 times more costly than just detecting the event.

In some other situations, reporting could be accomplished by writing into an allocated memory area. With such in-memory reporting, the detecting cost may be comparable to reporting cost. If detecting cost is close to reporting cost, it becomes impossible to lower the overhead by detecting all events and reporting only some of them. Approaches from section 4.2 need to be used then.

If detecting cost is not negligible, but still significantly cheaper than reporting cost, the base case algorithm can be applied with a minor modification of overhead calculation:

$$Overhead = 1 + (\sum_{i=1}^{n} F_i) * T_{detect} + (\sum_{i=1}^{n} F_{redi}) * T_{report}$$

$(\sum_{i=1}^{n} F_i) * T_{detect}$ cannot be changed – it is constant cost of detecting all events. So linear programming and continuous Knapsack solutions to the problem are still applicable.

4.3.5 *Minimal sampling frequency requirement*

In certain situations users want to capture at least some of events of an event class. For example, when building a visualization, users may want to see at least 1 event of the class in a 100 millisecond period, if this is the quantum of the visualization time scale. To achieve this the sampling frequency cannot not be lower than a certain limit. This is easy to achieve. In the probing solution we introduce an additional constraint that $F_{probing} \geq F_{minimal}$. This constraint does not change complexity of the problem.

In the base case solution we introduce a set of constraints $F_{redi} \geq F_{minimal\ i}$. Even with these constraints, the base case remains a linear programming problem.

4.3.6 *Selecting information weights*

Our solutions require users to choose the weights for each event class. This could be done in exploratory fashion in the base case, since the cost of solving the problem should be negligible when the expected number of data classes, n, is less than a hundred or so. In case of sampling via probing, this may not be viable for smaller number of n if $O(2^n)$ time algorithm is used.

If a user needs to collect all events of certain classes, tracing has to be used. This can be achieved by setting the information weight of this class to a very high value ensuring a solution that traces these events. Alternatively, class C_i can be removed from equations by assigning $F_{red\ i} = F_i$, $F_{istraced\ i} = F_i$, $F_{hits\ i} = F_i$, which are constants, and revising the remaining equations accordingly.

4.3.7 *Optimal approach for events occurring almost all the time*

In certain cases we can use heuristics instead of approximate algorithm to obtain good solutions. Increasing $F_{probing}$ may be a good strategy when $EventTime_i/EventPeriod_i$ ratios are close to 1. If all $EventTime_i/EventPeriod_i$ ratios are equal to 1, it means that all events occur all the time. For example, this may occur if we measure power consumption of independent hardware devices, which are active all the time, but with varying activity. This simplifies problem 2 to the following problem 3:

Problem 3.

Maximize the information value with the constraint $Overhead <= MaxAllowedOverhead$, where

$$InformationValue = \sum_{i=1}^{n} F_{hitsi} * W_i,$$

$$F_{hits\ i} = F_i, \text{if } C_i \text{is traced,} \quad (3)$$

$$F_{hits\ i} = F_{probing}, \text{if } C_i \text{is not traced, } \forall i = 1..n$$

It is easy to prove that the maximum in problem 3 is achieved by setting $F_{probing} = MaxF$ and all $F_{istraced\ i} = 0$.

Proof. Assume we have an optimal solution in which not all $F_{istraced\ i} = 0$. Consider any i such that class i is traced and $F_{istraced\ i} = F_i > 0$. We can increase the value of the objective function (3) by making this class not traced. We set increased $F_{probingnew} = F_{probing} + F_i$ and set $F_{istraced\ i} = 0$. The constraint 1 remains valid, since the first term increases by the same amount that the second term decreases. The new value of $F_{hits\ i} * W_i = (F_{probingnew} * W_i) = ((F_{probing} + F_i) * W_i)$ is larger or equal to the old value $(F_i * W_i)$. All other terms in the sum also increase or stay the same, so the value of the function (3) increases. This process can be repeated for every i such that $F_{istraced\ i} > 0$. At the end of the process we obtain a solution which is either better than the original – contradiction, or is equal to it, but all $F_{istraced\ i} = 0$. QED.

This shows that in case where events are happening a significant percentage of time, sampling is preferred solution and tracing should be done only if we have headroom for the overhead. On the other hand if the $EventTime_i/EventPeriod_i$ ratios are small – events are not happening most of the time - tracing becomes preferred solution.

5. APPLYING HYBRID PROFILING TO AN EXAMPLE USE CASE

To evaluate the applicability of the framework proposed above, we selected a mobile device profiling use case. The use case is realistic and occurs in real profiling of mobile devices, yet it is simple enough to describe here as an example. This use case has five event classes. These data classes represent tracing activity of file, window, kernel and font servers plus all other threads in the system (Table 1). Tracing all five data classes is impossible if the user wants to have 5% overhead, since the total frequency of traced events would be 1665 Hz (events/sec), while only 925 events per second can be traced with 5% overhead. I.e. our $MaxF = 925$. Tracing of all 5 classes produces about 10% overhead.

Entity	Execution time / Period ratio	Frequency (Hz)
File server	.30	359
Window server	.23	189
Kernel server	.09	153
Font server	.11	334
Others	.27	630
Total		1665

Table 1. Use case parameters

What can we do to maximize the information value and maintain 5% tracing overhead? We need to choose which event classes to trace and which to sample. This decision depends on the weights we assign to the different event classes. We use the formulas from Problem 2 to calculate the constraints and the information value. The optimal solution is found using an exhaustive search, since this is not prohibitively expensive for five data classes. If we had a much larger number of data classes, heuristics or an approximate method would have to be used.

If we assign every data class the same weight 1, the optimal solution is to sample everything and not trace any of the events. To test a variety of scenarios, we assume that file server events are more important than other events. If file server information weight is 2, while other weights remain 1, the optimal solution is to trace the file server, window server and font server events, while probing to get information about other events. In another scenario, users are not interested in the "other" events, so we raise the weights of four servers, while leaving the weight of "*others*" as 1. Now the system recommends tracing file server, window server and font server events, while sampling to get information about the rest. Finally, if the users want to see all information about kernel server, they could raise its weight and the system would propose to trace kernel server, file server and font server. All of these scenarios make sense and would be useful in real profiling situations. The example demonstrates that the system adapts the split between traced and sampled event classes according to user needs.

The tracing or sampling overhead for a single event also influences the suggestions for hybridization. If we assume that the event reporting cost is higher, so that MaxF is reduced to 641, the suggestions change. Now if we have file server information weight equal to 2 with other weights 1, the optimal solution is to trace the file server and window server. Font server events cannot be traced together with file and window server anymore.

We plan to develop a more user-friendly tool that would take the information weights and suggest top five different tracing-sampling combinations with their information values. As already can be seen from this example, the formal approach to hybrid data collection allows a fast and simple exploration of various tracing and sampling alternatives, sometimes yielding results that are not intuitive from the first glance. For example, we did not expect that by just raising the weight of file server information, we would get a suggestion to trace two other servers.

6. HYBRID APPROACH EVALUATION

The proposed hybrid approach for performance data acquisition in embedded software systems has the potential to limit the data collection overhead while providing partial completeness and causality.

It is important for user to select appropriate weights for different event classes. Such selection is domain and application specific.

Preceding sections provided some approaches to select the split between tracing and sampling using formal algorithms. Performance engineers who prefer to use simple heuristics could trace infrequent events and non-deterministic events that provide causality information, while sampling the rest. Such heuristic may provide less information though.

The hybrid approach also yields the following benefits:

- It can provide useful profiling results in shorter execution runs than can be provided by pure statistical sampling.
- It can be used to profile events that occur infrequently.
- It limits the profiling data volume, which makes storing, transfer and post processing easier. Performance engineers are more likely to make use of profilers if they are easy to use.
- It allows reconstructing the dynamic behavior of a software system.

The proposed hybrid approach also has some limitations:

- Trace and sampling instrumentation is required, which may alter the behavior of the original software system.
- It yields two separate sets of profiling data. These two sources of information need to be combined and synchronized during post-mortem analysis.

Certain information could be reconstructed from statistical samples gathered during an execution. Events that deterministically precede events captured in a sample could be added to the performance data. This direction needs to be explored in future research.

Event frequencies, periods and execution times of different event classes may not be very consistent and stable. A lot of performance engineers do not monitor classical real-time systems where events occur exactly periodically. Therefore execution time and period data obtained from one execution may not be exactly the same as in another execution. However, this does not negate the algorithms proposed. We believe that the algorithms will give a better estimate than a guess without any data. However, further research into dependency of the algorithms on the frequency precision is required. Also it may be possible to adjust the sampling and tracing frequencies dynamically depending on actual frequency of events during the test case execution.

7. RELATED WORK

We first proposed hybrid profiling in a short speculative position paper [10]. This research paper represents a substantial extension of our position, including new algorithmic approach for splitting the event classes into sampling and tracing subsets and example use case.

Jain [6] discusses performance data collection. He mentions "event-driven" (tracing) and sampling monitors. However, his only mention of hybrid monitoring concerns with

a mix of software and hardware monitors. Hybrid profiling, as we propose it, is not discussed in Jain's book.

Several tools exist for performance profiling of software systems. Many of these are sampling based profilers [1]. Some tools, such as Intel's Vtune [15], provide event tracing capabilities in addition to statistical sampling. However, the user cannot simultaneously use event tracing and statistical sampling during a single test run.

Hollingsworth et al. [5] developed a hybrid data collection approach that uses event tracing to record state transitions in counter and timer data structures. These structures are then sampled periodically to collect performance data. This is similar to our base case, however, Hollingsworth et al. do not address the question of what to trace and what to sample.

8. CONCLUSIONS

This paper describes a hybrid approach to the performance data collection. The hybrid approach involves striking a balance between event tracing and statistical sampling, combining the completeness of event tracing with low cost of statistical sampling. In addition, the proposed approach limits the profiling data volume. Useful profiling results can be obtained with relatively short execution runs.

The hybrid approach is sensitive to the choice of which performance data to collect using event tracing and which by statistical sampling. We present a formal approach for splitting event classes into the traced and sampled subsets.

We have presented the examples of a hybrid data collection approach for software execution time and resource consumption analyses. With a simple use case we demonstrated the power of the formal approach to maximize information amount in collected data without exceeding the expected overhead.

We believe that hybrid profiling should be incorporated in future profilers. It is likely that other dynamic analysis domains would also benefit from incorporating both complete and sampling based data collection.

9. ACKNOWLEDGMENTS

The authors want to thank Karel Driesen, Bil Lewis, Diana Lenceviciene, as well as anonymous reviewers for comments on the earlier versions of this paper.

10. REFERENCES

[1] J. Anderson, L. Berc, J. Dean, S. Ghemawat, M. Henzinger, S. Leung, R. Sites, M. Vandevoorde, C. Waldspurger, W. Weihl, Continuous Profiling: Where Have All the Cycles Gone?, *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997

[2] M. Arnold, B. Ryder, A Framework for Reducing the Cost of Instrumented Code, *Proceedings of the Conference on Programming Language Design and Implementation* (PLDI), 2001, pp. 168-179.

[3] V. Chvatal, *Linear Programming*, Freeman, 1983.

[4] Garey, M.R., and Johnson D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, New York, 1979.

[5] J. Hollingsworth, B. Miller, J. Cargille, Dynamic Program Instrumentation for Scalable Performance Tools, *Proceedings of the Scalable High Performance Computing Conference*, 1994

[6] Jain, R., *The Art of Computer Systems Performance Analysis*, Wiley 1991.

[7] R. Lencevicius, E. Metz, A. Ran; Software Validation using Power Profiles, *Proceedings of the 20th IASTED International Conference on Applied Informatics* (AI 2002), Feb 2002.

[8] Linear Programming Frequently Asked Questions, 2004 *http://www-unix.mcs.anl.gov/otc/Guide/faq/linear-programming-faq.html*

[9] E. Metz, R. Lencevicius, Efficient Instrumentation for Performance Profiling, *Proceedings of the 1st Workshop on Dynamic Analysis*, 2003, pp. 143–148.

[10] E. Metz, R. Lencevicius, Performance Data Collection: A Hybrid Approach, *Proceedings of the 2nd International Workshop on Dynamic Analysis*, 2004, pp. 48–51.

[11] S. Sahni, *Data Structures, Algorithms and Application in C++*, p. 697., Silicon Press, 2005.

[12] D. Stewart, Measuring Execution Time and Real-Time Performance, *Embedded Systems Conference (ESC)*, 2001.

[13] K. Subramaniam, M. Thazhuthaveetil, Effectiveness of Sampling Based Software Profilers, *1st International Conference on Reliability and Quality Assurance*, 1994, pp. 1–5.

[14] J. Vetter, D. Reed, Managing Performance Analysis with Dynamic Statistical Projection Pursuit, *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, 1999.

[15] Vtune Performance Analyzer, March 2004. *http://www.intel.com/software/products/vtune/.*

11. APPENDIX A: PROOF OF THEOREM 2

Theorem 2: The approximation algorithm generates a solution with objective function $\hat{f} > \frac{1}{2} f^*$, where f^* is the objective function value of an optimal solution.

Proof: Let OPT be an optimal solution, let $i_1 \le i_2 \le \ldots \le i_k$ be such that the only traced events are $C_{i_1}, C_{i_2}, ..., C_{i_k}$ in OPT, and let $T = \{ i_1, i_2, \ldots, i_k \}$.

Let $NT = \{ i_1, i_2, \ldots, i_n \} - T$ be the set of indices of the events that are sampled (i.e., not traced) in OPT. Let $\alpha = (\sum_{j=1}^{k} F_{hitsi_j}) / MaxF = (\sum_{j=1}^{k} F_{i_j}) / MaxF$, i.e., the fraction of MaxF taken up by the traced events in an optimal solution. Clearly, $\sum_{j=1}^{k} F_{i_j} = \alpha \; MaxF$ (1)

and $F_{probing} = (1-\alpha) MaxF$. (2)

If $k = 0$, then we know that OPT is $S_{0,0}$ and from the

algorithm we know that $\hat{f} \geq s_{0,0}$. Therefore, $\hat{f} = f^*$ and the theorem follows. On the other hand, when k = n, we know OPT is $S_{0,n}$. From the algorithm we know that $\hat{f} \geq s_{0,n}$. Therefore, $\hat{f} = f^*$ and the theorem follows.

So assume without loss of generality that $1 \leq k < n$. By definition we know that

$$f^* = \sum_{j \in T} F_j W_j + \sum_{j \in NT} F_{probing} \frac{EventTime_j}{EventPeriod_j} W_j .$$

Substituting equation (2) we know that

$$f^* = \sum_{j \in T} F_j W_j + (1-\alpha) MaxF \sum_{j \in NT} \frac{EventTime_j}{EventPeriod_j} W_j .$$

Substituting $\hat{f} \geq s_{0.0} = MaxF \sum_{j=1}^{n} \frac{EventTime_j}{EventPeriod_j} W_j$

and $\sum_{j=1}^{n} \frac{EventTime_j}{EventPeriod_j} W_j > \sum_{j \in NT} \frac{EventTime_j}{EventPeriod_j} W_j$

we know that $f^* < \sum_{j \in T} F_j W_j + (1-\alpha)\hat{f}$. (3)

When our approximation algorithm is generating the solution $S_{0,0}$, $S_{0,1}$, …, $S_{0,n}$, if all the events in T are traced in $S_{0,n}$, we know that $\sum_{j \in T} F_j W_j \leq s_{0,n} \leq \hat{f}$. So Equation 3 becomes $f^* < \hat{f} + (1-\alpha)\hat{f}$ and since $0 \leq \alpha \leq 1$, it then follows that $f^* < 2\hat{f}$ or $\hat{f} > 0.5 f^*$.

On the other hand when not all the events in T are traced in $S_{0,n}$, let C_{i_l} be the first event that is traced in OPT but not traced in solution S_{0,i_l}. Let R be the indices of the events traced in S_{0,i_l} that are different from the events C_j for $j \in T$.

Since C_{i_l} is *not* traced in S_{0,i_l} it must have been that if traced then all the previously traced events plus C_{i_l} would exceed MaxF, i.e., $\sum_{j=1}^{l-1} F_{i_j} + \sum_{j \in R} F_j + F_{i_l} > MaxF$. (4)

From $W_{i_j} \geq W_{i_{j+1}}$ for $1 \leq j < k$, $W_j \geq W_{i_l}$ for $j \in R$, and equations (1) and (4) we know that the per unit information value of the traced events in S_{0,i_l} together with C_{i_l} being also traced is larger than the one for the traced events in OPT. Therefore,

$$\frac{\sum_{j=1}^{l-1} F_{i_j} W_{i_j} + \sum_{j \in R} F_j W_j + F_{i_l} W_{i_l}}{\sum_{j=1}^{l-1} F_{i_j} + \sum_{j \in R} F_j + F_{i_l}} > \frac{\sum_{j=1}^{k} F_{i_j} W_{i_j}}{\alpha MaxF} .$$

Using equation (4) and simplifying the above inequality becomes

$$\sum_{j=1}^{l-1} F_{i_j} W_{i_j} + \sum_{j \in R} F_j W_j + F_{i_l} W_{i_l} > \frac{\sum_{j=1}^{k} F_{i_j} W_{i_j}}{\alpha} \quad (5)$$

We now show that $\hat{f} > \frac{\sum_{j=1}^{k} F_{i_j} W_{i_j}}{2\alpha}$. There are two cases we need to consider.

Case 1: $\sum_{j=1}^{l-1} F_{i_j} W_{i_j} + \sum_{j \in R} F_j W_j > F_{i_l} W_{i_l}$.

Substituting the conditions of the case in equation (5) we know that

$$2\left(\sum_{j=1}^{l-1} F_{i_j} W_{i_j} + \sum_{j \in R} F_j W_j\right) > \frac{\sum_{j=1}^{k} F_{i_j} W_{i_j}}{\alpha} .$$

Since $\hat{f} \geq \sum_{j=1}^{l-1} F_{i_j} W_{i_j} + \sum_{j \in R} F_j W_j$, it then follows that

$$\hat{f} > \frac{\sum_{j=1}^{k} F_{i_j} W_{i_j}}{2\alpha} .$$

Case 2: $\sum_{j=1}^{l-1} F_{i_j} W_{i_j} + \sum_{j \in R} F_j W_j \leq F_{i_l} W_{i_l}$.

Substituting the conditions of the case in equation (5) we know that $2F_{i_l} W_{i_l} > \frac{\sum_{j=1}^{k} F_{i_j} W_{i_j}}{\alpha}$.

Now consider solution $S_{i_l,0}$. Clearly, $\hat{f} \geq s_{i_l,0} \geq F_{i_l} W_{i_l}$. Substituting in the above inequality we know that $\hat{f} > \frac{\sum_{j=1}^{k} F_{i_j} W_{i_j}}{2\alpha}$.

Therefore in both of these cases we know that $\hat{f} > \frac{\sum_{j=1}^{k} F_{i_j} W_{i_j}}{2\alpha}$.

Substituting this in equation (3) we know that $f^* < 2\alpha\hat{f} + (1-\alpha)\hat{f}$.

Therefore, $f^* < (1+\alpha)\hat{f}$. Since $\alpha \leq 1$, we know that $f^* < 2\hat{f}$ or $\hat{f} > 0.5 f^*$. This completes the proof of the theorem. QED.