Automated Fingerprinting of Performance Pathologies Using Performance Monitoring Units (PMUs)

Wucherl Yoo, Kevin Larson, †Lee Baugh, Sangkyum Kim, Wonsun Ahn, Roy H. Campbell
{wyoo5, klarson5, kim71, dahn2, rhc}@illinois.edu, †lee.w.baugh@intel.com
Department of Computer Science, University of Illinois at Urbana-Champaign, †Intel Corp.

Abstract

Modern architectures provide access to many hardware performance events, which are capable of providing insight into architectural performance bottlenecks throughout the core and memory hierarchy. These events can provide programmers with unique and powerful insights into the causes of performance problems in their programs, but interpreting these events has been a significant challenge. We describe a technique that uses data mining to automatically fingerprint a program's performance problems, permitting programmers to reap the architectural insights made possible by the events while shielding them from the onerous task of interpreting raw events. We use a decision tree algorithm on a set of micro-benchmarks to construct a model of performance problems. This extracted model is able to divide a profiled application into program phases, and label the phases with the patterns of hardware bottlenecks. Our framework provides programmers with a detailed map of what to optimize in their code, sparing them the need to interpret raw events.

1 Introduction

Broadly speaking, program performance can be negatively affected by two phenomena: poor choice of algorithms and inefficient or ineffective use of hardware. Though both algorithmic and hardware-oriented performance tuning can be essential to achieve performance goals, many programmers overlook hardware-oriented performance tuning. This tendency is encouraged by a long standing and powerful trend of abstracting hardware: for example, where once programmers had to be keenly aware of their program's memory usage or of the underflow of their computations, now they can assume practically-unbounded virtual memory and significant accuracy even for very small numbers. Many of the underlying hardware's resource limitations or design tradeoffs are opaque by design, which can lead to suitable programs running correctly but suffering severe performance degradation from hardware bottlenecks. As a result of this opacity, identifying and resolving hardware-based *performance pathologies* (which we define as program-level behaviors that abuse bottlenecks in hardware) can involve a lot of guesswork and ad-hoc experimentation. The present trend towards multi-core machines with more elaborate interconnects and memory hierarchies only exacerbates this problem.

Microprocessor architects have responded to this problem by providing hardware performance events, such as those in Intel's Performance Monitoring Unit (PMU). These events can provide insights into how running code is exploiting hardware resources, and grant opportunities to discover bottlenecks in key architectural components like the core pipelines, execution units, memory hierarchy, and interconnects. Nevertheless, hardware performance events remain only rarely used by developers interested in performance tuning. We identify five major reasons for this: (1) **Lack of Standardization.** Different manufacturers, and even different product lines from the same manufacturer, offer significantly different events. (2) **Poor Validation.** Historically, hardware performance events are not well-validated. (3) **Inadequate Tools.** Historically, the tools ecosystem supporting Hardware-Performance-Monitoring-based performance tuning has been paltry for all but architectural experts. (4) **Poor Documentation.** Performance Monitoring events have not been well-documented, or have been documented at a level inaccessible to most users. (5) **Lack of Micro-architectural Knowledge.** Most importantly, significant micro-architectural understanding is generally required to interpret most events. For these reasons, raw hardware performance events do not provide an effective opportunity for performance tuning for most users.

In this paper, we propose a system that can automatically associate patterns of hardware performance events with program-level performance pathologies. These micro-benchmarks' executions are then profiled with hardware performance events, and the profiles are used to construct a decision tree. The patterns of specific hardware performance events during an interval are used to affix a pathology label to that interval's execution in the decision tree. Our mechanism relies on the notion of *hardware performance event fingerprints*. Suppose that during an interval of execution of some application, there is a hardware-borne performance pathology that significantly affects performance. This execution will consequently incur a significant increase of any hardware performance events associated with the pathology. We have observed that different executions that suffer from the same pathologies tend to see a respective increase in the same set of hardware performance events; we then call the set of events which thus mark a performance pathology as its *fingerprint*. In order to generate these fingerprints, we employ thirteen expert-generated micro-benchmarks; each micro-benchmark codifies and represents a single known pathology.

In order to perform the classification of workloads based on these fingerprints, we adapt a data-mining classifier, the decision tree. There are two phases for the classification: the *training phase* and the *profiling phase*. In the *training phase*, one decision tree is trained by the profiled measurements of hardware performance events generated by our micro-benchmarks. We profile the micro-benchmarks, collecting the hardware performance events identified as significant by CFS [6] and *gain ratio* [12]. We call the reduced set of the events *key events*. In addition, we split the profiled data into segments that represent durations of execution, which we call *time-slices*. We use the time-slices as aggregated points of data for the decision tree. In the *profiling phase*, we use the trained decision tree from the micro-benchmarks in order to detect performance pathologies exhibited by target applications. We profile target applications, collecting the *key events*, and bin the resulting data into time-slices. For each time-slice, we then use the decision tree to classify that time-slice's behavior against any of the pathologies which were used to train the decision tree. The underlying intuition is that similar patterns of events will occur if both the application and the micro-benchmark are limited by the same resource bottleneck.

2 Related Works

Modern processors contain hardware to monitor hardware performance events which can be used to provide detailed information to determine the performance of applications [4]. Oprofile [10] samples the hardware performance events, however it has a limited number of events that can be collected simultaneously. Similar to the work of Azimi *et al*. [1], Intel VTune Amplifier XE [9] supports multiplexing the performance monitoring hardwares, thus it can simultaneously collect an arbitrary number of the performance hardware events. Bitirgen *et al.* [3] present a framework that manages shared resources on chip multiprocessors. The framework uses a machine learning mechanism to formulate a predictive model of the resource allocation. Curtis-Maury *et al.* [5] propose an online performance prediction model via the identification of parallel application execution phases. Heath *et al.* [7] use the hardware performance events to manage thermal emergencies in server clusters by emulating temperature. Stoess *et al.* [16] present a power management framework using the hardware performance events in a virtualized environment. Schneider *et al.* [14] use the hardware performance events for adaptive optimizations in compilers and runtime systems. Shen *et al.* [15] use the hardware performance events to construct a model of requests of users to a concurrent server environment. Xu *et al.* [18] uses data mining to analyze console logs to detect anomalies in large-scale system. In this paper, we propose an automated system that fingerprints the pathological patterns of the hardware performance events and identifies the pathologies in applications. The automated discovery of pathologies will compliment previous research.

3 Design

3.1 Training Phase

Micro-Benchmarks	Pathology Description
Array / LL	Inefficient accesses on array / linked list incurring heavy cache misses over L2/L3/Memory resident-working set
Pointer	Pointer chasing accesses incurring heavy sequential cache misses over L2/L3/Memory-resident working set
BranchMis	Heavy branch misprediction
RS	Heavy usage in Reservation Station
FPU	Heavy software emulation of floating point instructions

Table 1: Description of Performance Pathologies Modeled by Micro-benchmarks

Micro-benchmarks

The *training phase* begins with the construction of *pathological micro-benchmarks*. Architecture experts are responsible for generating the micro-benchmarks that will be used to characterize known performance pathologies in the decision tree. Once the micro-benchmarks are constructed, we collect all the hardware performance events from a run of the micro-benchmarks. Since the hardware performance event collection mechanisms are separate from the main execution hardware, the

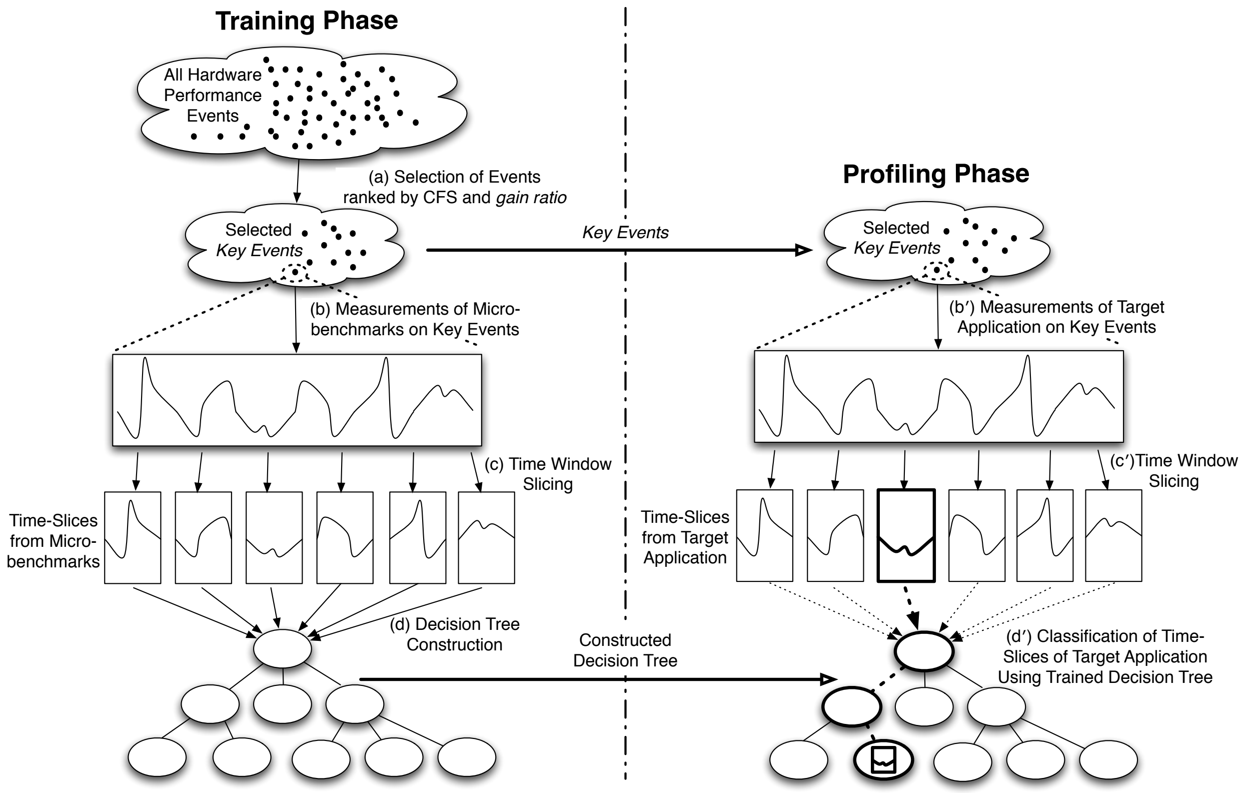
Training Phase
All Hardware Performance Events
(a) Selection of Events ranked by CFS and gain ratio
Selected Key Events
Key Events
Profiling Phase
Selected Key Events
(b) Measurements of Micro-benchmarks on Key Events
(b′) Measurements of Target Application on Key Events
(c) Time Window Slicing
(c′)Time Window Slicing
Time-Slices from Micro-benchmarks
Time-Slices from Target Application
(d) Decision Tree Construction
Constructed Decision Tree
(d′) Classification of Time-Slices of Target Application Using Trained Decision Tree

Figure 1: *The procedural flow.* (a) The micro-benchmarks are profiled to determine the most relevant events (*key events)*. (b) They are profiled again, collecting only the key events. (c) The profiled results are divided into time-slices. (d) The time-slices are used to construct a decision tree. (b′) To identify those pathologies in application code, the application code is also profiled with the set of relevant events. (c′) The measured events are divided into time-slices. (d′) Each time-slice is compared in the decision tree to identify the most relevant pathology.

events can be collected at nearly full execution speed, without the heavy overhead found in instrumentation mechanisms such as Pin [11].

Table 1 describes the micro-benchmarks. Each micro-benchmark features an initialization phase and an execution phase. The initialization phase sets up any data the execution may require and it is filtered out because of irrelevance to the pathology. The execution phase actually exhibits the pathology, and can be executed repetitively in order to provide needed execution duration. Each micro-benchmark is designed to codify at least one representative pathology as well as the *lack* of that pathology, while keeping execution as similar as possible. In the discussion below, we prepend `P_` to the bound of the micro-benchmarks (either `CPU` or `Mem`) configured to exhibit their pathology, and prepend `N_` to the bound configured *not* to exhibit the pathology. For example, the micro-benchmark, `Array` can be configured either to iterate over the working set in a random order (pathological) or a strided sequential order (non-pathological).

`Array`, `LL`, and `Pointer` are memory-bound micro-benchmarks representing pathologies in the memory hierarchy. By combining the pathologies from the memory-bound micro-benchmarks, `P_Mem` can model heavy cache misses on array data or linked-list data that are characterized by `Array` and `LL`. In addition, `P_Mem` models sequential cache misses raised by pointer chasing access patterns that are characterized by `Pointer`. Since applications may have varying memory footprints, even the same pathology can show different patterns of the hardware performance events depending on the data size. Thus, we collect a range of different pathologies from memory-bound micro-benchmarks by adjusting their parametrized working set size. For instance, `Array` can characterize at least three kinds of pathologies, each with its complement. We add a suffix, `L2`, `L3`, or `M` after the label `Mem` to indicate a working set resident in L2, L3, or Memory respectively; *i.e.*, `P_Mem_L3` for the memory-hierarchy pathology on L3-resident data.

`BranchMis`, `RS`, and `FPU` are CPU-bound micro-benchmarks representing the pathologies in core-pipeline or execution units of CPU. `P_CPU` models low prediction from branch predictor that characterized from `BranchMis`. In addition, `P_CPU` models heavy stalls from Reservation Station (RS) and heavy software emulation of floating point instructions instead of using Floating Point Units (FPU). Since CPU-bound micro-benchmarks are independent on the working set size, these micro-benchmarks do not have those additional labels about the working set size. Note that the labels given to pathologies (or non-pathologies) here will be used as classes of the classifications made by the decision tree; that is, execution of a profiled workload will be described as `P_Mem_L3` or `N_CPU`.

Key Events Selection

The dimension of the data space consists of hundreds of hardware performance events, most of which may be weakly correlated with the performance pathologies, thus irrelevant to classify the pathologies. Furthermore, some of the events may be highly inter-correlated with each other, thus they are redundant. Using all the events to train a decision tree might result in performance degradation due to either over-fitting the problem from redundant events or noise effects from irrelevant events. For this reason, we can yield a more accurate decision tree and decrease its training and classification time by only sampling the set of the events that are strongly correlated with the performance pathologies and weakly inter-correlated with each other.

We use CFS [6] and *gain ratio* [12] in order to select a set of the hardware performance events that achieves this goals. If we only pick the events with high *gain ratio*, we sometimes have highly inter-correlated events in the selected set of the events. Thus, we rank the events according to the heuristics of CFS to filter out the highly inter-correlated events. First, we choose the number of events that we want to reduce to. Using CFS, we select a subset of the events and calculate the ranking of the events in the subset. If there exist events not in the subset, they will be assigned the same lowest ranking. In addition, we also calculate rankings of the events according to *gain ratio*. We iteratively remove the event with the lowest average ranking of both analyses until we reach the threshold number of the events. In each iteration, we only need to use CFS to reselect the subset of the attributes and rank them as the *gain ratio* of the events remains the same.

Time Slicing

The performance monitoring hardware triggers an event when the number of the events overflow a certain threshold. Thus, the hardware performance events are easy to represent as aggregated count values. We aggregate all the measured events into separate *time-slices* that are the smallest units representing a phase in an application. This is important because applications sometimes consist of multiple phases that exhibit wildly varying behaviors. In order to detect changes in the phases, and to classify each phase into a pathology category, we aggregate the hardware performance events within time-slices instead of a single interval over the entire run. The time slicing mechanism increases the accuracy of the decision tree, not only by splitting the run into phases but also by providing more training points per phase. The aggregated events within a time-slice are normalized by dividing by the number of clock cycles in the time-slice. With this normalization, the event data collected from different time-slices can be treated as the same manner in the decision tree. It should be noted that not all instructions have lengths of one cycle, but because we are comparing ratios, multi-cycle instructions should have no effect on our results.

Our current implementation uses equal duration time-slices with equally spaced division points. We drop the last time-slice since it usually has a shorter duration than the other time-slices. The hardware performance events that are strongly correlated with a pathology will affect the aggregated events within the time slices. Thus, there is high probability that a dominant pathology makes noticeable patterns of the events and can be identified by our framework.

Decision Tree

We use the decision tree algorithm to identify the performance pathologies in applications. The decision tree algorithm constructs an analytic model that can fingerprint and classify the performance pathologies. Each leaf node is labeled as one of the pathology labels. Compared with other classification algorithms *e.g.*, Support Vector Machine (SVM) [17] and Neural Network [13], the decision tree algorithm has two merits: (1) The model constructed from the decision tree algorithm is easy to understand, which helps analyze the identified performance pathologies. (2) The decision tree has shorter classification times as well as shorter training times than the above classification algorithms. In our framework, the decision tree will not be changed, once it is constructed from the training set. End-users will not experience the offline training process of the classification algorithms, therefore long training times are irrelevant to end-users. However, end-users will use the constructed classifiers, the low classification times of the decision tree are preferable.

As shown in Figure 2, our framework constructs a decision tree that can classify the modeled pathologies. We reduce the number of the pathology classes by combining the time-slices from separate micro-programs. This helps to avoid the loss of accuracy of the decision tree by having too many classes. In addition, it is more resistant to the over-fitting problem. If we do not remove the

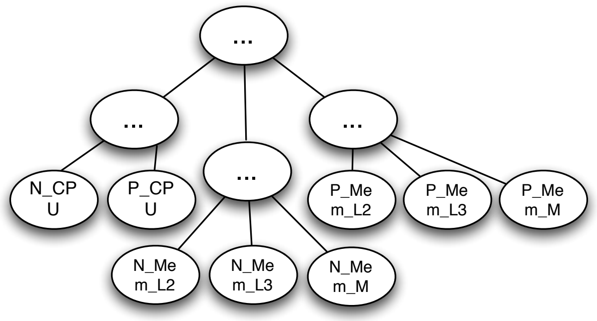

Figure 2: Abstract View of the Decision Tree.

names of the micro-programs in the pathology label, the decision tree identifies whether a time-slice resembles the time-slices from a specific micro-program instead of identifying a generalized pathology model. Our set of micro-benchmarks do not represent a complete set of pathologies. However, we believe that the identification coverage and accuracy of our decision tree can be improved by adding more expert-generated micro-benchmarks. For our experiments, we apply a popular decision tree algorithm: C4.5 [12].

3.2 Profiling Phase

The procedures of the *profiling phase* are shown in Figure 1.(b′)∼(d′). Users use the key events and the decision tree constructed in the *training phase*. Users collect the *key events* during profiling runs of a target application. The resulting profiles are divided into time-slices. The decision tree can classify the pathological patterns of the events in time-slices of target applications as one of the pathology labels. Classification of the time-slices is done by traversing a path of the decision tree from the root node to one of the leaf nodes. This leaf node has been assigned to one of pathology labels and it will assign the same label to the time-slice. The traversal of non-leaf nodes is determined by comparing a splitting value with one of the *key events*. Once the performance pathology is identified, the programmer can make adjustments in the code to mitigate the problem.

4 Evaluation

Our experiments are conducted on a machine with two 2.4 Ghz quad-core Intel Xeon E5620 CPUs, 12G of memory, 12M of shared L3 cache, and 1M of L2 cache. The hardware performance events (PMU events) are captured by Intel VTune Amplifier XE [9] as a kernel module compiled with Linux 2.6.35. The kernel module collects the user-level PMU events of the applications including the execution of the library modules. We use `gcc`, `g++`, and `gfortran` to compile the micro-benchmarks and the applications with optimization option "-O2". We select 40 of the PMU events as the *key events* by the ranking analysis in Section 3.1. The interval of a time-slice containing all the *multiplexing* groups is 400ms unless the CPU changes the base frequency. The working set sizes of the micro-benchmarks described in Table 1 range from 128KB to 256KB for L2-resident data, from 4MB to 8MB for L3-resident data, and from 128MB to 256MB for Memory-resident data. We experiment with the SPEC CPU2006 integer and floating-point benchmarks [8] and the PARSEC benchmarks [2] as target applications.

We perform 10-fold cross-validation to estimate the classification accuracy of the decision tree constructed in the *training phase*. Cross-validation is a broadly accepted technique used to estimate the classification accuracy of a decision tree. It can evaluate the accuracy of the decision tree which classifies the time-slices from micro-programs as the pathology labels. We also perform 10-fold cross-validation to estimate whether the time-slices from target applications embody sufficient information to identify unique characteristics of the applications. Note that this experiment is not used in either the *training phase* or the *profiling phase* in our framework, however this is useful to understand the characteristics of the target applications. First, we construct a decision tree with all the time-slices labeled as the application name from SPEC CPU2006 applications. Then, we conduct 10-fold cross-validation against the classifications of this decision tree to determine whether it can discriminate time-slices from the separate applications. This cross-validation result can help to understand the consistency of the patterns in the PMU events among time-slices which share the same label. If the accuracy of the decision tree is particularly low on a label, this may imply the existence of inconsistencies between the time-slices with that label. That application may include multiple patterns or phases that cause poor prediction from the decision tree.

Benchmarks	Accuracy	*CC*	*IC*
Micro-benchmarks	99.96%	30939	13
SPEC	97.74%	45932	1049
PARSEC	99.40%	52213	317

Table 2: 10-fold Cross Validation against a Decision Tree

Table 2 shows the classification accuracy of the trained decision tree trained from the micro-benchmarks, SPEC, and PARSEC. We denote *CC* to be the number of correctly classified time-slices, *IC* to be the number of incorrectly classified time-slices, and the *accuracy* to be $\frac{CC}{CC+IC}$. Note that these decision trees are different from those built during the profiling phase. The classification accuracy of the micro-benchmarks are slightly better than that of the applications of SPEC CPU2006 and PARSEC. This result is expected since the micro-benchmarks are designed to have consistent program phases. The accuracy result from the applications of SPEC CPU2006 and PARSEC shows that our mechanism extracts sufficient information to classify the time-slices from separate applications.

We identified one following pathology using the classified results. 1917 out of 2822 time-slices from the *canneal* benchmark are classified as `pM`. The *canneal* application was shown to exhibit the pathologies associated with inefficient pointer accesses. Investigation showed that when *canneal* read in data elements from the input data set, it called a function `create_elem_if_necessary()` that searched a map for an element sharing the same name. If no element existed, it would create a new element and add that element's name to the map. It called `create_elem_if_necessary()` for every element in the input data set.

5 Conclusion

In order to identify performance pathologies, we propose an automated framework that uses micro-benchmarks developed by architecture experts to model the performance pathologies by exhibiting event patterns for resource bottlenecks. Our framework selects a refined set of the hardware performance events that are highly correlated with the performance pathologies. A decision tree trained with the selected events models fingerprints of the characteristics of the performance pathologies. The decision tree is able to identify the performance pathologies in a target application from these fingerprints. With the PMU *multiplexing* support and a sufficient set of hardware performance events, our mechanisms are applicable to any architectures. We envision future work encompassing more architectures with different ISA.

References

[1] Azimi, R., Stumm, M., and Wisniewski, R. Online performance analysis by statistical sampling of microprocessor performance counters. In *Proceedings of the Annual International Conference on Supercomputing* (2005), ACM, pp. 101–110.

[2] Bagrodia, R., Meyer, R., Takai, M., Chen, Y., Zeng, X., Martin, J., and Song, H. Parsec: A parallel simulation environment for complex systems. *Computer 31*, 10 (1998), 77–85.

[3] Bitirgen, R., Ipek, E., and Martinez, J. Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture* (2008), IEEE Computer Society, pp. 318–329.

[4] Browne, S., Dongarra, J., Garner, N., London, K., and Mucci, P. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the ACM/IEEE conference on Supercomputing* (Washington, DC, USA, 2000), IEEE Computer Society.

[5] Curtis-Maury, M., Dzierwa, J., Antonopoulos, C., and Nikolopoulos, D. Online power-performance adaptation of multithreaded programs using hardware event-based prediction. In *Proceedings of the Annual International Conference on Supercomputing* (2006), ACM, pp. 157–166.

[6] Hall, M. A. Correlation-based feature selection for discrete and numeric class machine learning. In *Proceedings of the International Conference on Machine Learning* (San Francisco, CA, USA, 2000), pp. 359–366.

[7] Heath, T., Centeno, A., George, P., Ramos, L., Jaluria, Y., and Bianchini, R. Mercury and freon: temperature emulation and management for server systems. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems* (2006), ACM, pp. 106–116.

[8] Henning, J. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News 34*, 4 (2006), 1–17.

[9] Intel. Vtune Amplifier XE. *Web site: http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe* (2011).

[10] Levon, J., and Elie, P. Oprofile: A system profiler for linux. *Web site: http://oprofile.sourceforge.net* (2011).

[11] Luk, C., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V., and Hazelwood, K. Pin: building customized program analysis tools with dynamic instrumentation. In *ACM SIGPLAN Notices* (2005), vol. 40, ACM, pp. 190–200.

[12] Quinlan, J. *C4. 5: programs for machine learning*. Morgan Kaufmann, 1993.

[13] Rumelhart, D. *Learning internal representations by error propagation*. MIT Press, 1986.

[14] Schneider, F., Payer, M., and Gross, T. Online optimizations driven by hardware performance monitoring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (2007), ACM, pp. 373–382.

[15] Shen, K., Zhong, M., Dwarkadas, S., Li, C., Stewart, C., and Zhang, X. Hardware counter driven on-the-fly request signatures. In *ACM SIGARCH Computer Architecture News* (2008), vol. 36, ACM, pp. 189–200.

[16] Stoess, J., Lang, C., and Bellosa, F. Energy management for hypervisor-based virtual machines. In *Proceedings of the USENIX Annual Technical Conference* (2007), USENIX Association, p. 1.

[17] Vapnik, V. An overview of statistical learning theory. *Neural Networks, IEEE Transactions on 10*, 5 (1999), 988–999.

[18] Xu, W., Huang, L., Fox, A., Patterson, D., and Jordan, M. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM Symposium on Operating Systems Principles* (2009), ACM, pp. 117–132.