CHAPTER 11

Machine Learning in Action: Examples

A breakthrough in machine learning would be worth ten Microsofts.

—Bill Gates

Machine learning is an important means of synthesizing and interpreting the underlying relationship between data patterns and proactive optimization tasks. Machine learning exploits the power of generalization, which is an inherent and essential component of concept formation through human learning. The learning process constructs a *knowledge base* that is hardened by critical feedback to improve performance. The knowledge base system gathers a collection of facts and processes them through an inference engine that uses rules and logic to deduce new facts or inconsistencies.

As more and more data are expressed digitally in an unstructured form, new computing models are being explored to process that data in a meaningful way. These computing models synthesize the knowledge embedded in the unstructured data and learn domain-specific trends and attributes. More sophisticated models can facilitate decision support systems, using hierarchies of domains and respective domain-specific models. Machine learning plays an important role in automating, expanding, and concentrating procedures for unearthing learnings in ways that traditional statistical methods are hard-pressed to match.

This chapter presents examples in which machine learning is used as the principal constituent of a feedback control system. We discuss machine learning usage in areas related to datacenter workload fingerprinting, datacenter resource allocation, and intrusion detection in ad hoc networks. These examples demonstrate an intelligent feedback control system based on the principles of machine learning. Such systems can enable automated detection, optimization, correction, and tuning throughout high-availability environments, while facilitating smart decisions. Furthermore, these systems evolve and train themselves, according to platform needs, emerging use cases, and the maturity of the knowledge data available in the ecosystem. The goal is to create models that act as expert systems and that automatically perform proactive actions that can later be reviewed or modified, if necessary.

A traditional system uses a collection of attributes that determine a property or behavior in current time. An expert system uses machine learning to discover the nature of the change resulting from the learning process and analyze the reasoning behind better adaptation of the process. Such a system can be either history determined or state -determined. A state-determined system can be described in terms of transitions between states in consecutive time intervals (such as first-order Markov chains). The new state is uniquely determined by the previous state. To adapt, the organism, guided by information from the environment, must manage its essential variables, forcing them to operate within the proper limits by manipulating the environment (through the organism's motor control of it), such that it then acts on the variables appropriately.

Figure 11-1 illustrates an autonomic system that is constructed by using modular functions to enact an intelligent feedback control system. Machine learning plays a significant role in modeling the knowledge function, which is used to store the rules, constraints, and patterns in a structured manner. New knowledge is synthesized, using the elements of existing structures and new learnings. The collective knowledge enacts a feedback control loop, which enables a stable and viable system. The supporting functions that facilitate an intelligent feedback control system are as follows:

- A *sensor function* to sense any changes in the internal or external environment (such as component temperature, power, utilization, and aberrant behavior).
- A *motor function* to compensate for the effects of environmental disturbances by changing the system elements, thereby maintaining equilibrium.
- An *analytical function* to analyze the sensor channel data to ascertain if any of the essential variables are operating within viable bounds, or limits.
- A *planning function* to determine the changes that need be made to the current system behavior to bring the system back to the equilibrium state in the new environment.
- A *knowledge function* that contains the set of possible behaviors that can be applied to the new environment. The planning tool uses this knowledge to select the appropriate action to nullify the disturbance; the motor channel applies the selected behavior. The knowledge function is synthesized, using the generalization process, which can be an ongoing task that effectively develops a richer hypothesis space, based on new data applied to the existing model.

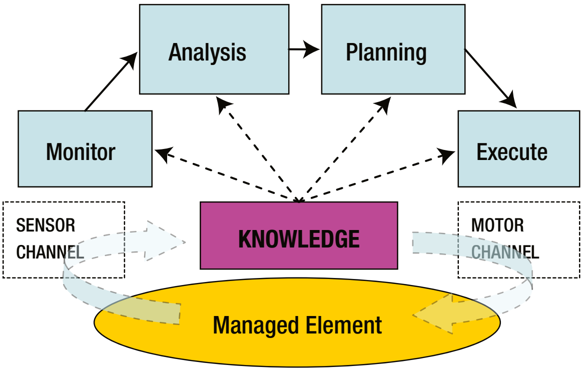


Figure 11-1. *Modular functions as fundamental elements for building autonomics architecture*

These functions enable key elements to model a practical system by using an abstracted cybernetic description (regulation theory). This description abstracts a set of interrelated objects that can receive, store, process, and exchange data. Cybernetic systems can be represented by means of control objects and control systems. Control systems transmit the control information to the controlled objects via sets of effectors. The state information on a controlled object is received through a set of receptors and is transmitted back to the control system, thereby creating a feedback loop. This feedback loop is capable of developing an autonomic system that operates an effective control system through adaptive regulation. The model comprises the following five necessary and sufficient interacting subsystems, which, collectively, constitute an organizational structure that affords system viability:

- *Infrastructure* to interact with the operational environment, which is controlled by the management process.
- *Coordination* to promote the dissemination of policy data that allow collaboration and coordination.
- *Control* for intervention rules as well as policy adherence, resource compliance, accessibilities, and responsibilities.
- *Intelligence* for planning ahead in anticipation of changes in the external environment and capabilities. Intelligence aids in capturing a complete view of the system environment and benefits the system in formulating alternate strategies, which are necessary for adapting to changing conditions to keep the system viable.
- *Policy* to steer the organization toward a purposeful goal by formulating policy functions that lead to planning activities.

Viable System Modeling

Controlled systems may demonstrate a high degree of dynamism in their interactions that may result in unpredictable behaviors. Viable system modeling facilitates a framework that allows coordination, coevolution, and survivability, using monitoring, control, and communication abstractions. Such modeling helps the system survive in a constantly changing and unpredictable environment; the modeling is built to outlast external stresses and demands variability, and it adapts to any unexpected stimuli. You abstract the framework in such a manner that variability in the usage model does not interfere with stabilizing the system within its viable limits. Unpredictable behavior, intermittent failures, and scattered knowledge generate a sizeable uncertainty, which can cause reactive or suboptimal decisions as well as rendering ineffective traditional programming paradigms, which operate on the principals of independence and static behavioral models. Programming strategies need to be built to characterize and optimize runtime patterns, using dynamic policies. This requires autonomous instantiation of mechanisms that respond to changing dynamics. Additionally, programming models must take into account the separation between policy management, activation mechanisms, computation, and runtime adaptation. Isolation can be realized using abstractions that make it possible to hide the implementation choices. Abstractions establish a common view of a component model to allow interoperability via communication semantics.

Intelligent feedback control implementation facilitates adaptation, which lets the controlled process change its configuration over time by dynamically adapting to specific needs and requirements. Adaptation is typically triggered by the rule engine, owing to faulting, or changing behavior, of a resource in platform, and is achieved through changes in the set of resources, components, or dependencies. A necessary condition for adaptation is the preservation of the existing semantics, with an ability to reconfigure and adapt to the new environment. This is supported by implementing adding, plugging, and unplugging component controllers dynamically, thereby adding or removing the functional aspect of a component. The two categories of adaptation are as follows:

- *Functional adaptation* adapts the architectural behavior of the component to new requirements or environments.
- *Nonfunctional adaptation* adapts the nonfunctional architecture of the container to the new requirements or environment (e.g., changes in security policy or communication protocol).

Adaptation functions require the ability to identify dynamically changing patterns and behaviors with these properties:

- The ability to evaluate when and how much. Heuristics are built to identify appropriate conditions that demand reconfiguration. Reactive reconfiguration or tuning can easily lead to oscillation. Additionally, the amount of reconfiguration depends on the application-specific optimization function (also called the *cost function*), which is maximized for a given policy.
- A weighted cost function to evaluate the significance of specific objectives in cases in which multiple objective functions race for adaptability.
- Low-latency evaluation and optimization of the cost function to satisfy the handling of control function in real time.
- The ability to identify and resolve resource conflicts and oscillations resulting from competing objectives.
- The ability to log resource attributes and behavioral patterns to aid in establishing a rational reasoning process for future optimizations and conflict resolutions.

Knowledge synthesis through pattern identification plays a pivotal role, modeling several behavioral trends that recur over time. These patterns are appealing for proactive analysis, which paves the way for monitoring and analyzing the process by focusing on the most significant activities. Pattern analysis is commonly used to determine workflow behavior as well as the correspondence between related actions. Pattern analysis is also used to predict the choices that are more likely to yield the desired policy compliance, thereby supporting the selection of the best action to be activated from among a set of possible candidates. The framework execution model is responsible for collecting two types of information: *raw data* logged during the execution of workload and *processed data* derived from the raw data, which are synthesized to describe behavioral patterns.

Figure 11-2 conceptualizes an adaptation framework for a *server system*, in which subfunctional components are isolated into discrete and independent blocks. The *monitor function* monitors the performance parameters and corresponding resource utilization. This is logged into the knowledge base, which traces the relationship between resource utilization and various performance attributes. The *drift detector* evaluates the deviation between measured performance and desired performance. A deviation (positive or negative) triggers a correction to the cost function as well as to the resource control block. Whereas the cost function manipulates the relative coefficients of the fitness equation, the resource control mechanism reevaluates resource allocation, based on rational reasoning and fuzzy logic. This process continues until all the objectives are met.

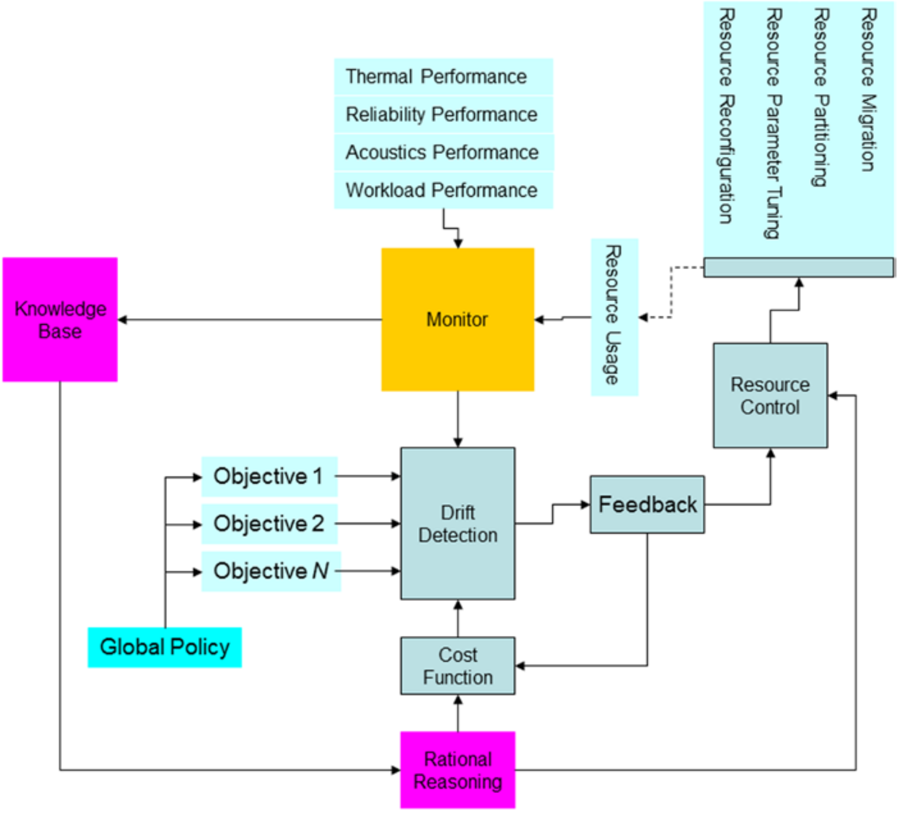


Figure 11-2. *Adaptation function of a server system; based on the feedback, the cost function block reevaluates the function weights, whereas the resource control block reevaluates resource allocation*

Example 1: Workload Fingerprinting on a Compute Node

Datacenter power consumption has increased drastically in recent years, and controlling the power intake of servers has become critical. To develop efficient servers, server platforms require online characterization to determine platform parameters for the tuning required for various system features because of the complex dependencies among them. Feedback-directed optimization of servers with insights gained from workload observation has proven to be more effective than static configurations. The server processor and chipsets expose software-configurable margins and range limits, called *control parameters*, which can be tuned to achieve a balance between power and performance. Some of these parameters are set by the platform's basic input/output system (BIOS) once at boot time and remain static throughout. One-time configuration at boot time renders the system nonresponsive to load variation.

The example given here describes a dynamic characterization technique using machine learning algorithms that determine tuned values from runtime program phases. A self-correcting workload fingerprint codebook accelerates phase prediction to achieve continuous and proactive tuning. Parameters such as memory and processor power states can be set dynamically, based on observed demand variations, by the operating system or the hardware modules. Autonomous systems then trade off system margins to gain power or performance advantages, depending on the use case. Proactive tuning prepares the system to tune itself in advance and avoids the response lag characteristic of reactive systems. Various machine learning techniques, such as clustering, classifiers, and discrete phase predictors, are applied to the data collected from subsystems of the processor. Additionally, it is crucial to ascertain the appropriate algorithms and operations, while considering extrapolative efficiency at the least computational cost.

Performance-monitoring units facilitate measurement of fine-grained events through hardware counters. These counters allow application profilers to reveal the application's time-varying phase behaviors that repeat in a defined pattern over their lifetime. A program phase can be described as a discontinuity in the time in which observable characteristics vary distinctively enough to effect an equally measurable system impact variation. The phase characteristics and probabilistic sequence of known phases are represented as a fingerprint. Fingerprints facilitate proactive models for load balancing, thermal control, and resource allocation within a collection of servers in a datacenter. The steps employed in the learning process can be summarized as follows:

1. Execute the workload in the server system(s).
2. Identify relevant process control parameters, and relate them to process goals (workload throughput, server power, thermal variance, and so on).
3. Synthesize the attributes of a phase and phase sequence.
4. Build a phase prediction model to forecast future behavior of the workload.
5. Tune the system control parameters proactively.

Workload fingerprinting allows proactive self-tuning, which avoids a lag between the feedback and the control (or the reactive control) by learning the underlying relationships between operational workload phases and corresponding application behavior, thereby facilitating dynamic adaptation in changing environments. Understanding application behavior has also been a key source of insight for driving several new architectural features, such as built-in memory power control, thermal throttling, turbo boosting, processor cache size adaptation, and link configurations. The feature-specific controls that constitute the decision space are dynamically tuned to the current and future phases through a multiobjective coordinated tuning process to achieve globally optimal results. Traditionally, the decision space of a process is defined by control parameters, which are tuned, based on insights derived from offline empirical analysis of data collected by running well-known benchmarks and establishing the average case. These control parameters are then statically programmed at boot time by the system BIOS. A better approach is to program dynamically the parametric values synthesized by proactive simulation, using a machine learning method that exploits an intelligent feedback-based self-tuning process.

Phase Determination

It is quite common to employ well-understood benchmark tools as models that approximate real-world workload behavior, because they have a finite completion time, while exhibiting unique resource usage patterns. A program phase has been defined and extended in many ways by researchers, based on the goals to be achieved. This example defines a *program phase* as a variable time interval during execution in which a set of observable characteristics exhibits spatial uniformity and distinctiveness. The example uses a multivariable phase determination technique consisting of multiple dominant variables that are both externally observable and related to the platform control variable. The measured variables m_i are the values of performance events obtained from hardware counters when running a range of bootstrap workloads. The elements of this initial dataset $M\ (m_i \mid m_i \in M, i \le N)$ serve as the building blocks of the phase model.

The motivation to choose N variables to be measured comes from the objective of the study. The feature selection process is commonly applied to large datasets to filter out correlations and vastly reduce the computational complexity of algorithms in subsequent stages. To improve classifier accuracy and efficiency, special tools, such as the *correlation-based feature selection* (CFS) algorithm, are applied to reduce the dataset's dimensionality. CFS is a best-first search heuristic algorithm that ranks the worth of subsets rather than individual features. The algorithm filters out the features that are effective in predicting the class, along with the level of intercorrelation among features in the subset, by calculating the feature–class and feature–feature matrices. The features selected by CFS exhibit a high degree of correlation to the reference class, but redundant variables are removed. Given a subset *S* consisting of *k* features, the heuristic score is given as

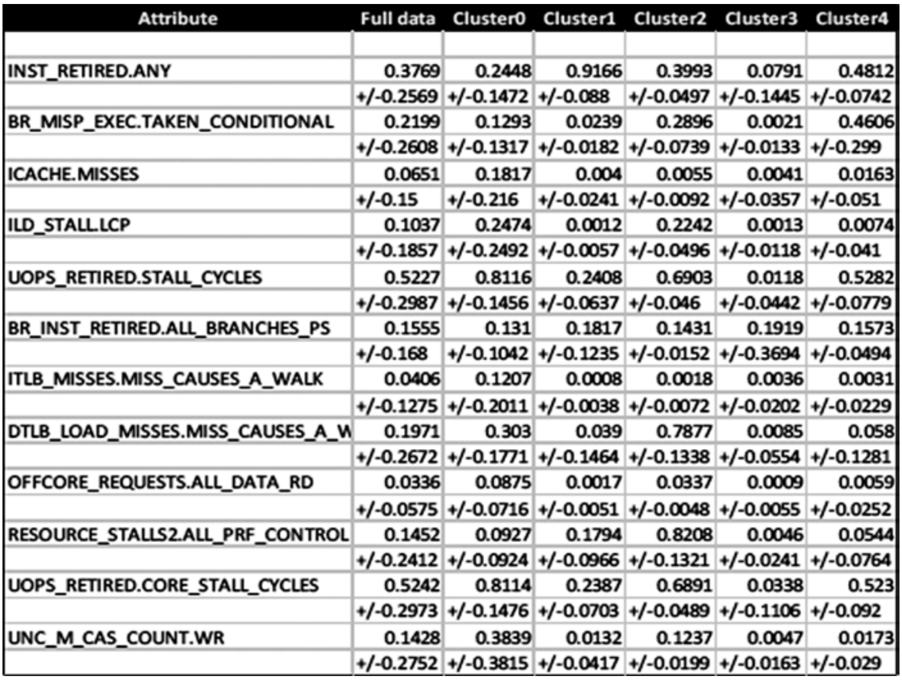
$$Merit_{s_k} = \frac{k\overline{r_{cf}}}{\sqrt{k + k(k+1)\overline{r_{ff}}}}, \tag{11-1}$$

where $\overline{r_{cf}}$ is the average feature–class correlation, and $\overline{r_{f_f}}$ is the average feature–feature intercorrelation.

CFS transforms the measured variable set *M* to a reduced variable set *D*, yielding $(d_j \mid d_j \in D, j \le \overline{N})$, *D* = *CFS*(*M*), and $\overline{N} \le N$. The next step is to build the phase model by using the selected features and applying the simple *k*-means algorithm to group the uncorrelated variable instances. The *k*-means algorithm is an unsupervised machine learning algorithm that partitions the input set into *k* clusters, such that each observation belongs to a cluster with the nearest mean (the Euclidean distance). The objective function of the *k*-means algorithm can be represented as

$$J = \sum_{l=1}^{k} \sum_{m=1}^{j} \| d_m^{(l)} - c_l \|^2, \tag{11-2}$$

where c_l is the chosen centroid of cluster *l* and $d_m^{(l)}$ represents *m*th datapoint in *l*th cluster. You consider each cluster a phase *ϕ*. This example, with *k* = 5, produces a model that has the mean and standard deviation of the 12 filtered variables (see Figure 11-3).



Attribute	Full data	Cluster0	Cluster1	Cluster2	Cluster3	Cluster4
INST_RETIRED.ANY	0.3769	0.2448	0.9166	0.3993	0.0791	0.4812
	+/-0.2569	+/-0.1472	+/-0.088	+/-0.0497	+/-0.1445	+/-0.0742
BR_MISP_EXEC.TAKEN_CONDITIONAL	0.2199	0.1293	0.0239	0.2896	0.0021	0.4606
	+/-0.2608	+/-0.1317	+/-0.0182	+/-0.0739	+/-0.0133	+/-0.299
ICACHE.MISSES	0.0651	0.1817	0.004	0.0055	0.0041	0.0163
	+/-0.15	+/-0.216	+/-0.0241	+/-0.0092	+/-0.0357	+/-0.051
ILD_STALL.LCP	0.1037	0.2474	0.0012	0.2242	0.0013	0.0074
	+/-0.1857	+/-0.2492	+/-0.0057	+/-0.0496	+/-0.0118	+/-0.041
UOPS_RETIRED.STALL_CYCLES	0.5227	0.8116	0.2408	0.6903	0.0118	0.5282
	+/-0.2987	+/-0.1456	+/-0.0637	+/-0.046	+/-0.0442	+/-0.0779
BR_INST_RETIRED.ALL_BRANCHES_PS	0.1555	0.131	0.1817	0.1431	0.1919	0.1573
	+/-0.168	+/-0.1042	+/-0.1235	+/-0.0152	+/-0.3694	+/-0.0494
ITLB_MISSES.MISS_CAUSES_A_WALK	0.0406	0.1207	0.0008	0.0018	0.0036	0.0031
	+/-0.1275	+/-0.2011	+/-0.0038	+/-0.0072	+/-0.0202	+/-0.0229
DTLB_LOAD_MISSES.MISS_CAUSES_A_W	0.1971	0.303	0.039	0.7877	0.0085	0.058
	+/-0.2672	+/-0.1771	+/-0.1464	+/-0.1338	+/-0.0554	+/-0.1281
OFFCORE_REQUESTS.ALL_DATA_RD	0.0336	0.0875	0.0017	0.0337	0.0009	0.0059
	+/-0.0575	+/-0.0716	+/-0.0051	+/-0.0048	+/-0.0055	+/-0.0252
RESOURCE_STALLS2.ALL_PRF_CONTROL	0.1452	0.0927	0.1794	0.8208	0.0046	0.0544
	+/-0.2412	+/-0.0924	+/-0.0966	+/-0.1321	+/-0.0241	+/-0.0764
UOPS_RETIRED.CORE_STALL_CYCLES	0.5242	0.8114	0.2387	0.6891	0.0338	0.523
	+/-0.2973	+/-0.1476	+/-0.0703	+/-0.0489	+/-0.1106	+/-0.092
UNC_M_CAS_COUNT.WR	0.1428	0.3839	0.0132	0.1237	0.0047	0.0173
	+/-0.2752	+/-0.3815	+/-0.0417	+/-0.0199	+/-0.0163	+/-0.029

Figure 11-3. *Phase model: cluster mean and standard deviation*

Once the phase model is trained, it can be tested against subsequent runs of the workload data. A classifier algorithm from a related class of machine learning algorithms can generate a tree, a set of rules, or a probability model to identify quickly the correct phase. One example is the tree representation obtained from a decision tree classifier algorithm upon training with the cluster data, as illustrated in Figure 11-4. This classifier achieves approximately 99 percent accuracy in phase identification.

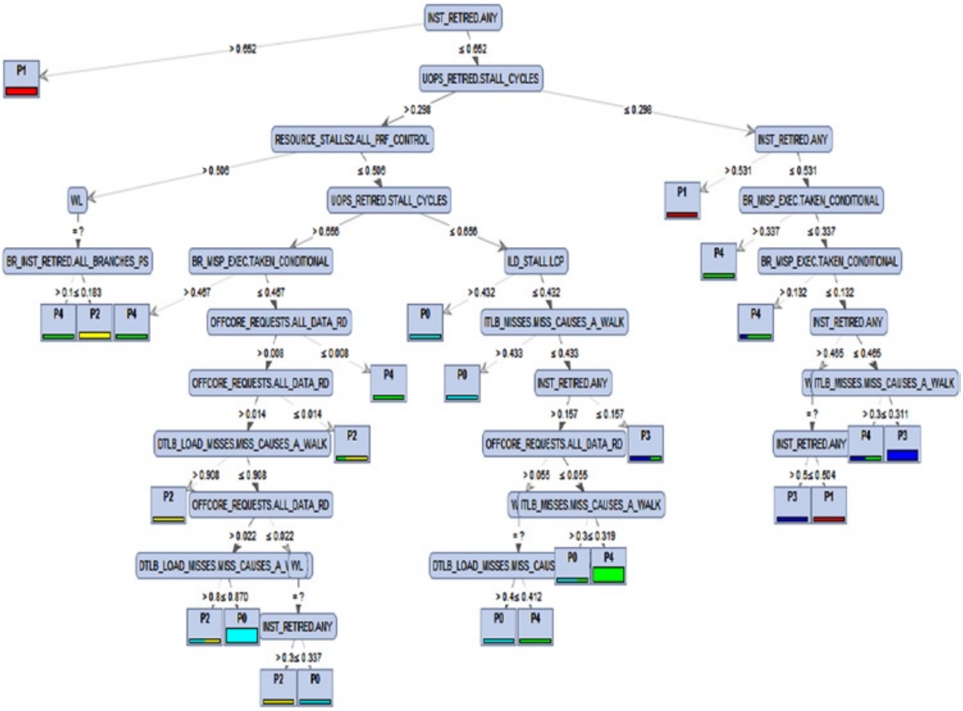


Figure 11-4. *Decision tree representation of the model*

The initial training data set of 37 workloads results in five clusters, with each cluster representing a workload phase (see Figure 11-5). In each phase the 12 dominant features display a diverse variation pattern, with at least one feature being the primary predictor. As mentioned earlier, a phase transition can be identified by the classifier tree, using the feature-specific threshold values.

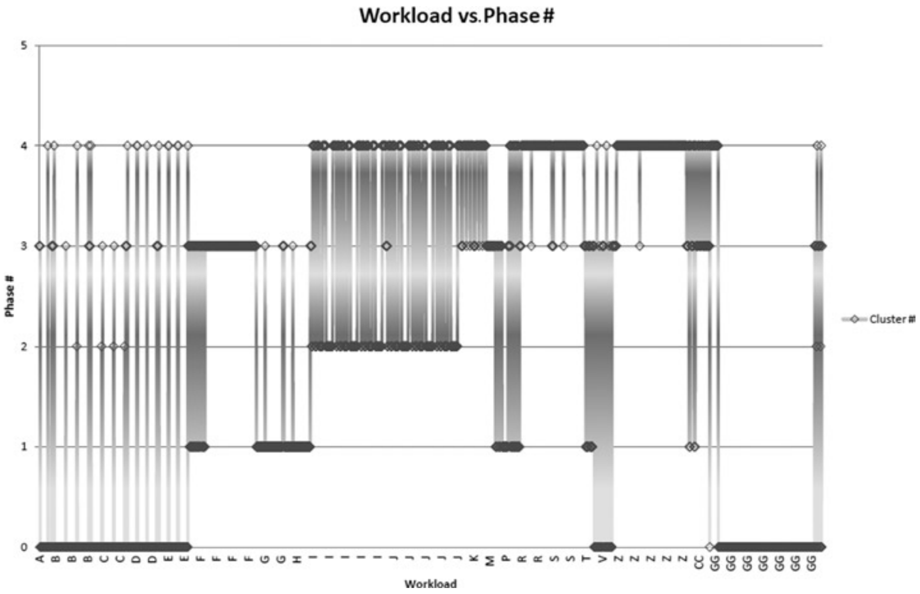


Figure 11-5. *Phase transition diagram for various workloads*

Figure 11-6 shows the workload–phase boundaries, with seven clusters found in four workloads. Each workload is characterized by a unique composition of workload phases. Once the workload is identified, it is phase characterized by its resource utilization and time series pattern. Whereas workload 3 consists mainly of phase 7, workload 1 is a complex mix of phases 3, 5, and 6. These phases discover the characteristic that is unique to a workload instance at any given time.

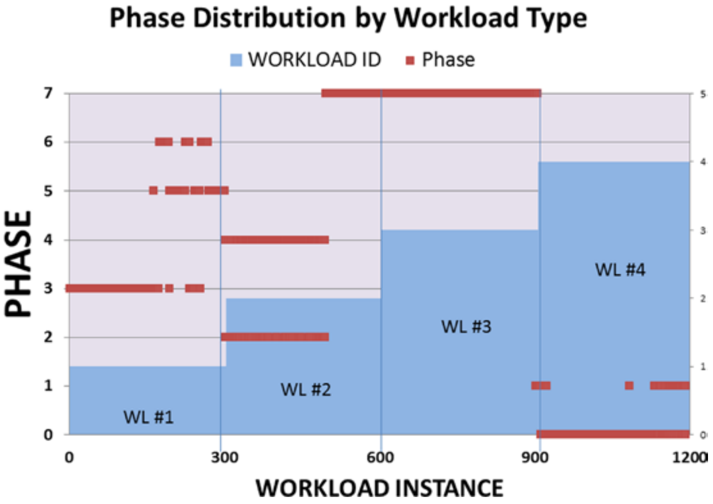


Figure 11-6. *Workload and phase dependency graph; a workload may share phase characteristics with other workloads and can cater to one or more phases*

Fingerprinting

Workloads undergo phases of execution, while operating under multiple constraints. These constraints are related to power consumption, heat generation, and *quality of service* (QoS) requirements. Optimal system operation involves complex choices, owing to a variety of degrees of freedom, with respect to power and performance parameter tuning. The process involves modeling methodology, implementation choices, and dynamic tuning. Fingerprinting acts as an essential feature that captures time-varying behavior of dynamically adaptable systems. This ability is used as a statistical output that aids in reconfiguring hardware and software ahead of variation in demand and that enables the reuse of trained models for recurring phases. Pattern detection also assists in predicting future phases during the execution of workloads, which prevents reactive response to changes in workload behavior.

As part of the *workload fingerprinting* process (see Figure 11-7), individual performance characteristics are collected at a given interval, classified, and aggregated to establish patterns representative of an existing workload or collection of workloads. System control agents, such as I/O schedulers, power distribution, and dynamic random access memory (DRAM) page policy settings, can use this information to tune their parameters or schedule workloads in real time. Fingerprinting can roughly be attributed using three properties: size, phase, and pattern. The machine learning process facilitates synthesis of these properties by measuring or data mining performance characteristics over a finite period. Generally, the feature selection process allows automatic correlation of performance matrices with occurrences of unique workload behaviors, thereby aiding in speedy diagnosis and proactive tuning. Fingerprinting data can be combined with simple statistical functions, such as optimization, visualization, or control theory, to create powerful operator tools. Furthermore, fingerprints help contain prolonged violation of one or more specified *service-level objectives* (SLOs), which involves performing proactive actions to return the system to an SLO-compliant state.

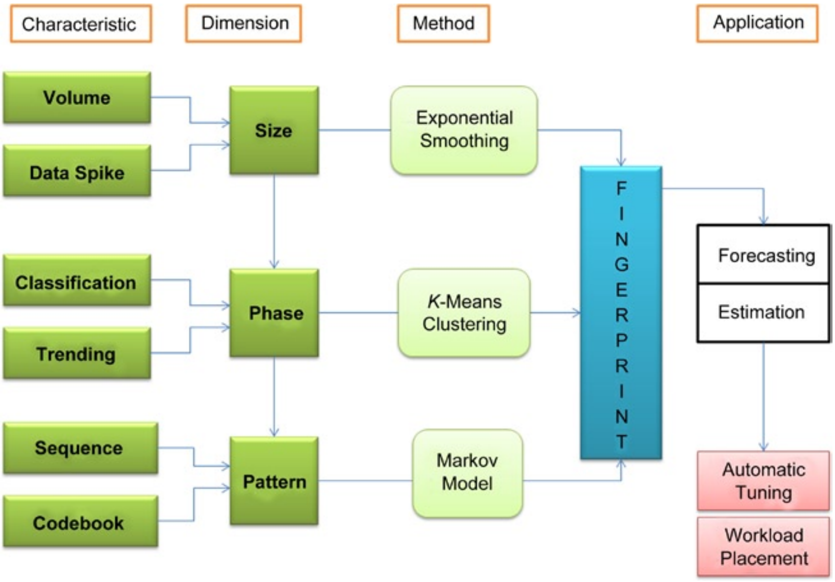


Figure 11-7. *Workload fingerprint: a quantifiable form of characterization*

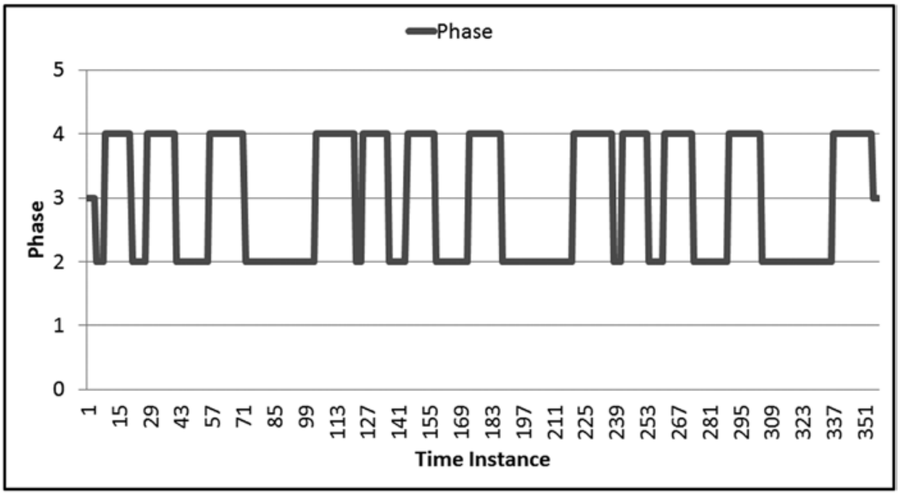
Size Attribute

The *size* attribute is useful in proactive provisioning of resources. Using the size dimension provides answers to the following questions:

- What is the shape of the distribution? Which resources are more popular than others? Is there a significant tail?
- What is the spatial locality in accesses to the groups of popular objects during spikes?

Phase Attribute

A *phase* represents a unique property that characterizes the behavior of an ongoing process. In this example the phase demonstrates unique power, temperature, and performance characteristics. As described previously (see the section "Phase Determination"), this example employs a simple *k*-means algorithm to synthesize exclusive behaviors in the form of clusters, represented as phases. This process is executed once all the relevant feature vectors are identified. These vectors are observations that directly or indirectly reflect the unique behavior in the form of resource usage. The phases are compressed representative output that can be used in conjunction with any other statistical parameter for prediction of behavior. Sequences of phases can be seen as patterns. Patterns represent a unique time-varying characteristic of the workload. Figure 11-8a illustrates the phase sequence during one execution of a workload. Figure 11-8b displays the encoded representation of the phase sequence.



(a)

```
{3:3,2:3,4:11,2:6,4:12,2:14,4:15,2:31,4:17,2:2,4:11,2:7,4:12,2:14,
4:14,2:31,4:17,2:3,4:11,2:6,4:12,2:15,4:14,2:31,4:17}
```

(b)

Phase #

Phase #	0	1	2	3	4	5	6	7
0	0.943775	0.056225	0	0	0	0	0	0
1	0.384615	0.589744	0	0	0	0	0	0.025641
2	0	0	0.5	0	0.482143	0	0	0.017857
3	0	0	0	0.972826	0	0.016304	0.01087	0
4	0	0	0.201439	0	0.798561	0	0	0
5	0	0	0	0.026667	0.013333	0.946667	0.013333	0
6	0	0	0	0.04878	0	0.02439	0.926829	0
7	0	0.004902	0	0	0	0	0	0.995098

Workload #	0	1	2	3	4	5	6	7
1	0	0	0	0.613333	0	0.25	0.136667	0
2	0	0	0.186667	0	0.463333	0	0	0.35
3	0	0	0	0	0	0	0	1
4	0.856164	0.133562	0	0	0	0	0	0.010274

(c)

Figure 11-8. *(a) Test workload phases, (b) run-length encoded phase sequence, and (c) phase transition likelihood matrix and workload-phase dependency matrix; the workload 1 phase dependency is highlighted*

Pattern Attribute

A *pattern* is defined as a sequence of phases that repeats. Once a sequence is identified, it can be used to predict future phases and the duration of current phases. In the time series operations, you construct the vocabulary for the time series pattern database. Each alphabet in the vocabulary is represented by the operating phase of the workload. This phase is measured in a fixed interval of length (T). The pattern matrix can be represented as M_{ij}, where i represents the pattern, and j stands for the frequency of that pattern. As new patterns are identified, they are updated into the pattern matrix, and old and infrequent patterns are deprecated. You can use a discrete time Markov chain (DTMC) (see Equation 11-3) to identify underlying patterns in a time series sequence of changing phases,

$$\mathbb{P}(q_{t+1} = S_j \mid q_t = S_i, q_{t-2} = S_k, \cdots) = \mathbb{P}(q_{t+1} = S_j \mid q_t = S_i), \tag{11-3}$$

where q_t represents the current state, and S_i represents one of the phases of operation. In this model a state transition (phase change) follows the Markov property and then creates transitions between states (phases), based on a learned model for forecasting. You can use a moving window to monitor real-time data and produce an autoregressive model for recently observed data, which is then matched to the state of the learned Markov model. The model also makes corrections, if necessary, to adapt to the changes. By tracing the time series progression from one phase to another, you can build a transition function of the Markov model (see Figure 11-8c).

Forecasting

The workload forecasting module detects trends in the workload and makes predictions about future workload volume. If the target workload demonstrates a strong periodic behavior, a historic forecast can be incorporated into workload forecasting. This allows the policy decision to react proactively to the workload spikes ahead of time. This also helps you take advantage of the heterogeneous compute and I/O resources offered by cloud computing providers. Furthermore, based on the extracted patterns, you may distribute the workloads in a manner that creates different performance models.

Example 2: Dynamic Energy Allocation

Controlling the amount of power drawn by server machines has become increasingly important in recent years. The accuracy and agility of three types of action are critical in power governance:

- Selecting which hardware elements must run at what rates to meet the performance needs of the software
- Assessing how much power must be expended to achieve those rates
- Adjusting the power outlay in response to shifts in computing demand

Observing how variations in a workload affect the power drawn by different server components provides data critical for analysis and for building models relating QoS expectations to power consumption. This next example describes a process of observation, modeling, and course correction in achieving autonomic power control on an Intel Xeon server machine meeting varying response time and throughput demands during the execution of a database query workload. The process starts with fine-grained power performance observations permitted by a distributed set of physical and logical sensors in the system. These observations are used to train models for various phases of the workload. Once trained, system power, throughput, and latency models participate in optimization heuristics that redistribute the power to maximize the overall performance per watt of the server.

The term *power optimization* denotes the act of targeting and achieving high levels of power-normalized performance at the application level. For a software application, such as a business transaction service or content retrieval service, the significant performance metrics include the number of requests serviced (throughput) and the turnaround delay (response time) per request. Optimizing power entails multiple dynamic tradeoffs. Typically, a system can be represented as a set of components whose cooperative interaction produces useful work. These components may be heterogeneous or presented with heterogeneous loads, and they may vary in their power consumption and power control mechanisms. At the level of any component—such as a processing unit or a storage unit—power needs to be increased or decreased on an ongoing basis, according to whether that component's speed plays a critical role in the overall speed or rate of execution of programs. In particular, different application phases may have different sensitivities to component speed. For instance, a memory-bound execution phase will be less affected by central processing unit (CPU) frequency scaling than a CPU-bound execution phase. With the execution reordering that most modern processors employ, the degree to which a program benefits from out-of-order execution varies from one phase to another. Moreover, the rate at which new work arrives in a system changes, and, as a result, the overall speed at which programs have to execute to meet a given service-level expectation varies with time. Thus, the needed power performance tradeoffs have to occur on a continuous basis.

Arguably, given the self-correcting and self-regulating aspects common in systems today, software-driven power performance should be unnecessary. For example, in power control algorithms, CPUs and DRAMs transition into lower frequencies or ultralow power modes during low-activity periods. Although circuit-level self-regulation is highly beneficial in transitioning components to low-power states, software needs to wield policy control over which activity should be reduced, and when, to facilitate the transition of hardware into power-saving modes.

Harnessing power savings on less busy servers is a delicate task that is hard to delegate to hardware-based recipes. Servers are typically configured for handling high rates of incoming work requests at the lowest possible latencies. Therefore, it is not uncommon for servers to have many CPUs and a large amount of physical memory over which computations and data remain widely distributed during both high- and low-demand periods. Owing to the distributed nature of activities, slowing down a single CPU or DRAM can have unpredictable performance ramifications; it can be counterproductive to push part of a server into ultralow power operation. At the other extreme, when power approaches saturation levels, hardware is ill positioned to determine or enforce decisions about which software activities can tolerate reduced performance and which must continue as before.

Thus, software must share with hardware the responsibility of determining when and in which component power can be saved. Here, we consider an autonomic solution for fine-grained control over power performance tradeoffs for server configurations. The solution consists of ingredients for observing, analyzing, planning, and controlling the dynamic expenditure of power in pursuance of an application-level performance objective that is specified as an SLO. This solution uses a time-varying database query workload, in which the learning machine simultaneously changes the power allocation to CPUs and DRAM and gathers performance and power readings via a set of distributed physical and logical sensors in the server. Through these observations, models are trained for various phases of the workload. Based on these models, the optimization heuristic redistributes the power to maximize the overall performance per watt of the server. Experimental measurements demonstrate that a heuristic improves performance and power, as needed or as permitted by the performance objective.

Learning Process: Feature Selection

The primary role of a learning process is to identify relationships between the total *power expended* (P) and two measures of performance: *response time* (R) and *throughput* (T). These relationships are synthesized as models, and optimization techniques use these models to achieve better power performance efficiency.

The process of generalization requires a classifier that inputs a vector of discrete feature vectors and outputs an operating phase, which can be summarized as follows:

- Fine-grained and time-aligned component-level power readings at multiple power rails of the primary components (CPU and dual inline memory module [DIMM])
- System-level readings corresponding to three quantities, each averaged over a small time interval: (1) *P*, the total system power; (2) *T*, the application-level throughput; and (3) *R*, the response time experienced by requests

The component-level power readings are aligned with the {*P*, *T*, *R*} tuples. This entire data collection is then used to divide the {*P*, *T*, *R*} space into classes (phases). Within each class or phase a linear function can relate *P*, *T*, and *R* to the component-level power readings. These linear relationships are used in optimization planning, whose objective may be to minimize *P* (total system power) or maximize *T* (application-level throughput), subject to *R*'s (response time) not exceeding a specific threshold. Learning continues online. Therefore, as the workload evolves, the models and optimization planning adapt.

The *support vector machine* (SVM) technique may be employed to divide the {*P*, *T*, *R*} space into different phases and to obtain linear relationships governing the {*P*, *T*, *R*} variables in each phase. As discussed previously, SVM is a computationally efficient and powerful technique; invented by Boser, Guyon, and Vapnik (1992), it is employed for classification and regression in a wide variety of machine learning problems. Given a data collection relating a set of training inputs to outputs, an SVM is a mathematical entity that accomplishes these tasks:

1. The SVM describes a hyperplane (in some higher dimension) whose projection into the input space separates inputs into equivalence classes, such that the inputs in a given class have a linear function mapping them to outputs that is distinctive for that class.
2. The hyperplane whose projection is the SVM maximizes the distance that separates it from the nearest samples from each of the classes, thus maximizing the distances between classes, subject to a softness margin.
3. The SVM creates a softness margin that permits a bounded classification error, whereby a small fraction of the inputs that should be placed on one side of the projection are instead placed within a bounded distance on the other side (and are therefore misclassified); this margin allows a pragmatic tradeoff between having a high degree of separation between classes (i.e., better distinctiveness) and having too many outliers.

Equation 11-4 expresses each element of {*P*, *T*, *R*} as a linear function of the five power readings per processor within each given class or phase. Whereas (V_{CPU}) yields power going into the processor, V_{DIMM} measures power in memory modules that are connected to and controlled from the processor. The variable *J* represents a given class, $\{P_J(t),R_J(t),T_J(t)\}$ represents a tuple from a sample numbered *t* in the training set, and the various power readings associated with that sample are represented by $V^*(t)$. The phases *J*; constants K^*_P, K^*_R, K^*_T; and coefficients α^*_* and β^*_* are all estimated through the SVM regression technique.

$$\underline{\text{CPU Power Readings (2)}} \qquad \underline{\text{Memory Power Readings (4)}}$$

$$\begin{aligned}
P_j(t) - K_P^J &= \sum_{CPU=i} \alpha_{PK}^{iJ} V_{CPU}^i(t) + \sum_{DRAM\,CH=i} \beta_P^{iJ} V_{DIMM}^i(t) \\
R_j(t) - K_R^J &= \sum_{CPU=i} \alpha_{RK}^{iJ} V_{CPU}^i(t) + \sum_{DRAM\,CH=i} \beta_R^{iJ} V_{DIMM}^i(t) \\
T_j(t) - K_T^J &= \sum_{CPU=i} \alpha_{TK}^{iJ} V_{CPU}^i(t) + \sum_{DRAM\,CH=i} \beta_T^{iJ} V_{DIMM}^i(t)
\end{aligned} \tag{11-4}$$

Learning Process: Optimization Planning

Energy and performance models have a number of degrees of freedom and conflicting objectives that are difficult to optimize collectively. For example, consider the following objectives:

- Best performance per watt
- Staying within a power limit
- Response time ≤ a service-level agreement (SLA) threshold

Conflicts can manifest themselves among these objectives, with considerations such as

- How to obtain a given throughput within a *system power budget*
- How to obtain a given throughput under a *response time threshold*

In the common case, P (total system power) is affected by both performance targets—throughput and response time. Also in the general case, performance is influenced by the power spent in both processors and DIMM modules. Thus, optimization planning must grapple with meeting a compound objective: one in which power expended toward one objective generally comes at the cost of another. Once the coefficients of the linear estimation model for power, throughput, and response time are synthesized, these models can be used as a synthetic feedback in a multiobjective optimization through a feedback control loop. This example uses an *adaptive weighted genetic algorithm* (AWGA) method to search for the global optimal in a scenario with multiple goals. In this machine learning technique a successful outcome is defined as one that redistributes power in such a way that power, response time, and the reciprocal of throughput all meet the viable limits. More generally, a set of fitness functions $\{f_n\}$, one per objective n, determines the optimality of a candidate setting (i.e., a vector describing the distribution of power among components) for each of the objectives. In AWGA, for a population ϕ of candidate settings $\{\mathbf{x}\}$, $F_n^{max} = \max(f_n(x) \mid x \in \phi))$ and $F_n^{min} = \min(f_n(\boldsymbol{x}) | \boldsymbol{x} \in \phi)$, you compute, respectively, the fitness bounds for each of a set of $n = 1, 2, \ldots, N$ objectives, where each x in ϕ is a vector whose fitness function represents a feasible power distribution among components, such as CPUs and DIMMs. You may then choose an N objective fitness function F that evaluates an aggregate fitness value. For example, in the case of AWGA, F can be chosen as

$$F = \sum_{n=1}^{N} \frac{f_n(\mathrm{x}) - F_n^{min}}{N \cdot (F_n^{max} - F_n^{min})}. \tag{11-5}$$

An *evolutionary algorithm* (EA) selects parents from a given generation of ϕ (usually employing an elitism process that allows the best solution[s] from the current generation to carry over unaltered to the next), from which to produce power-feasible offspring as new candidates for the next generation. In the objective space, F_n^{min} and F_n^{max} represent extreme points that are renewed at each generation. As the extreme points, fitness bounds $\{(F_n^{min}, F_n^{max}) | n = 1, 2, \ldots, N\}$ are renewed at each generation, and the contribution (weight) of each objective is also renewed accordingly.

Learning Process: Monitoring

Achieving power-efficient performance and abiding by power and performance constraints call for real-time feedback control. An autonomic system implements continuous feedback-based course correction, with the following provisions:

- *Monitoring* infrastructure to sample or quantify physical and logical metrics, such as power, temperature, and activity rates, and to obtain statistical moments of the metrics
- *Analysis* modules to distill relationships between monitored quantities (e.g., between power, temperature, and performance) and to determine whether one or more operational objectives are at risk
- A *planning* element to formulate a course of action, such as suspending, resuming, speeding up, or slowing down various parts of a system, to effect a specific policy choice (e.g., to limit power or energy consumed or to improve performance)
- A capability to *execute* the formulated plan and thereby *control* the operation of the system

Usually, a knowledge base supplements analysis and planning. The knowledge base may be an information repository that catalogs the allowable actions in each system state, or it may be implicit in the logic of the analysis, planning, and control capabilities. In a system designed for extensibility, the knowledge base typically incorporates an adaptive mechanism that tracks and learns from prior decisions and outcomes. For intelligent feedback control processes, fine-grained and lightly intrusive power performance monitoring is a key element of the adaptive power management infrastructure. The ideal monitoring mechanism operates in real time (i.e., reports data that are as recent as possible) and is not subject to the behavior(s) being monitored. In this configuration logical sensors at the operating system and software levels offer a near real-time information stream consisting of rates at which common system calls, storage accesses, and network transfers proceed. These logical sensors are supplemented with power sensing through physical sensors.

A *telemetry bus* is used to collect data from physical (hardware) and logical (software) sensors and send them to a monitoring agent. In particular, power sensing is accomplished by sensing *voltage regulator* (VR) outputs at each processor chip. The *monitoring agent,* to which the telemetry data are sent, processes the data, organizes them as a temporally aligned stream of power and performance statistics, and transmits the stream to a remote machine for further storage or analysis. The monitoring infrastructure provides the ability to obtain distinct power readings for each processor. Each processor controls distinct memory channels, with multiple DIMMs per channel; each pair of memory channels furnishes one V_{DDQ} signal; and summing those V_{DDQ} readings gives the power expended in the memory subsystem for each processor. The data collected by these sensors are refined through a succession of transformations (see Figure 11-9):

- *Sensor hardware abstraction (SHA) layer*: This layer interacts with the sensors and communication channels. It uses adaptive sampling, such that measurements are only as frequent as necessary, and it eliminates redundancies.
- *Platform sensor analyzer*: This layer removes noise and isolates trends, which makes it easier to incorporate recent and historical data as inputs in further processing.
- *Platform sensor abstraction*: This layer provides a programming interface for flexible handling of analyzed sensor data through the control procedures implemented above it.
- *Platform sensor event generation*: This layer makes it possible to generate signals. Signals facilitate event-based conversations from control procedures, thereby allowing further control to be hosted in a distributed set of containers (such as local or remote controller software and operating system modules). The prior successive refinements bridge the gap between the raw data that sensors produce and the processed, orderly stream of performance and power readings and alerts that software modules can receive and analyze further.

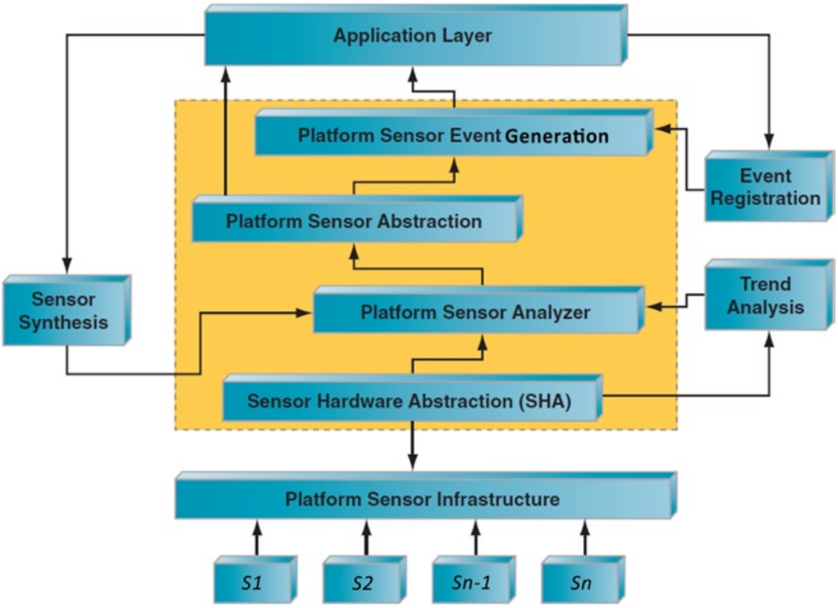

Application Layer
Platform Sensor Event Generation
Event Registration
Platform Sensor Abstraction
Sensor Synthesis
Platform Sensor Analyzer
Trend Analysis
Sensor Hardware Abstraction (SHA)
Platform Sensor Infrastructure
S1
S2
Sn-1
Sn

Figure 11-9. *Sensor network model: sensor network layered architecture (S1, S2, . . . , Sn) represents platform sensors (CPU/DIMM power, thermal, performance, and so on). Source: A Vision for Platform Autonomy: Robust Frameworks for Systems (Intel, 2011)*

Although a machine can be readily furnished with a metered power supply to sense total power, an instrumentation capability that yields the fine-grained decomposition of power requires nontrivial effort. Moreover, adding many physical power sensors in production machines is neither necessary nor practical, in terms of cost. Event-counting capabilities in modern machines offer a potent alternative means of estimating component power when direct measurement is not practical. One simple yet accurate way of estimating the power draw for recent CPUs is to project it on the basis of usage and power-state residencies, using trained models. Such training can be made more accurate by including execution profiles that capture what fraction of the instructions falls into each of a small set of categories, such as single instruction, multiple data (SIMD); load/store; and arithmetic logical unit (ALU). DRAM power can similarly be estimated on the basis of cache miss counts, or DRAM operations that are counted at the memory controllers and tracked through processor event monitors. DRAM power estimation permits measurement of DRAM energy at DIMM granularity with sufficient accuracy to enable efficient control of DRAM power states. Efficient control of DRAM energy allows us not only to reduce the cost of hardware infrastructure, but also to improve energy efficiency by reducing the guard bands required to compensate for underprediction. Furthermore, overprediction can also be reduced to avoid performance degradation.

Decision space that facilitates optimal distribution of power among competing components is obtained by process control methods, in which privileged software can modify its power draw. The first method, which is commonly used in Intel-based processors, is to change the P-states and C-states (Siddha 2007). The second method is to change the average power level, using a control known as running average power limit (RAPL) capability for CPUs and DRAM modules. CPU RAPL provides interfaces for setting a power budget for a certain time window and letting the hardware meet the energy targets. Specifying the power limit as an average over a time window allows us to represent physical power and thermal constraints. Privileged software can use the RAPL capability by programming to an interface register the desired average level of power to which the hardware can guide the processor via its own corrective frequency adjustments over a programmable control window. The window size and power limit are selected, such that, at either a single-machine level or a datacenter level, correction to a machine's power is driven quickly. In practice the window size can vary between milliseconds and seconds—the former to satisfy power delivery constraints, the latter to manage thermal constraints. The RAPL concept extends to memory systems as well, aided by the integration of the memory controller into each multicore processor in several recent versions of Intel platforms. Although CPU and memory energy can be regulated individually, it is possible to build a coordinated self-tuning approach, in which power regulation is part of a joint optimization function supported by the machine learning technique discussed in the following section.

Model Training: Procedure and Evaluation

For the model training the data collection module collects time-aligned readings from the power-monitoring sensors. Additionally, it gathers response times and requests completion rates from a database performance module. These readings provide the input–output training vectors $\{P_*(t), R_*(t), T_*(t), \text{and } V^i_{CPU}(t),\}$ (see Equation 11-4). The training data are obtained through a cross-product of two sets of variations:

- *Variation of demand:* This parameter controls how long each of a number of threads in the workload driver waits between completion of a previous request and issuance of a new request.

- *Variation of supply*: This control varies the CPU and memory RAPL settings, thereby varying the supply of power to CPU and DRAM.

In this example the workload uses time-varying think time varying from 0 to 100. For each think time, CPU RAPL limits are varied between 20W and 95W. SVM model training on the basis of these data is then used to categorize the data into distinct phases (J), following which the SVM model parameters for each phase $\{K^J_P, K^J_R, K^J_T, \alpha^{ij}_P, \alpha^{ij}_R, \alpha^{ij}_T, \beta^{ij}_P, \beta^{ij}_R, \beta^{ij}_T\}$ are evaluated.

The SVM-based classification yields decomposition into three phases, as shown in Figure 11-10.

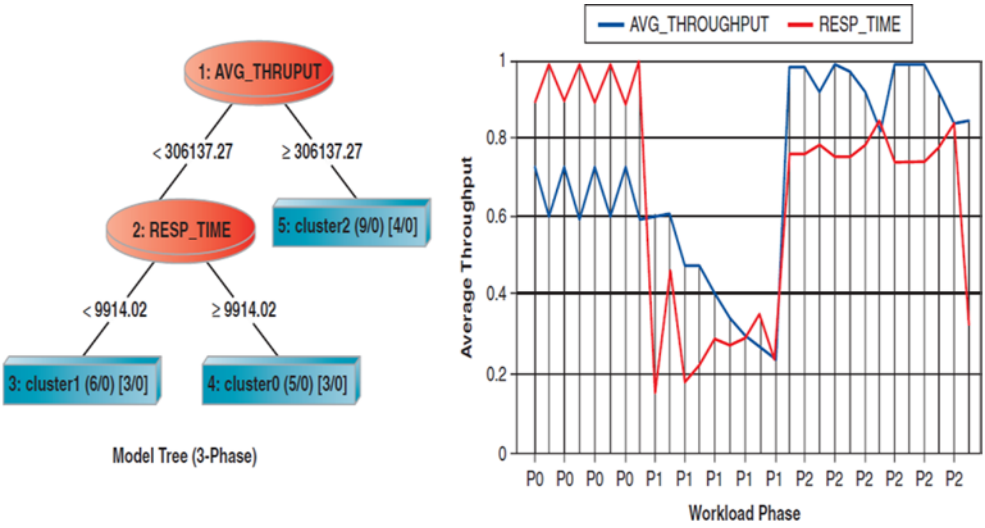


Figure 11-10. *Model tree depicting three phases (P0, P1, P2) in a workload characterized by throughput and response time*

Accordingly, three different sets of modeling parameters (i.e., for $J = 0, 1, 2$) in Equation 11-4 relate CPU RAPL parameters to total system power, throughput, and response time outcomes. Figure 11-10 demonstrates how the total wall power estimated on the basis of the RAPL parameters in Equation 11-4 compares with that actually measured. Figures 11-11 and 11-12 illustrate the close agreement between estimated and measured results from the training.

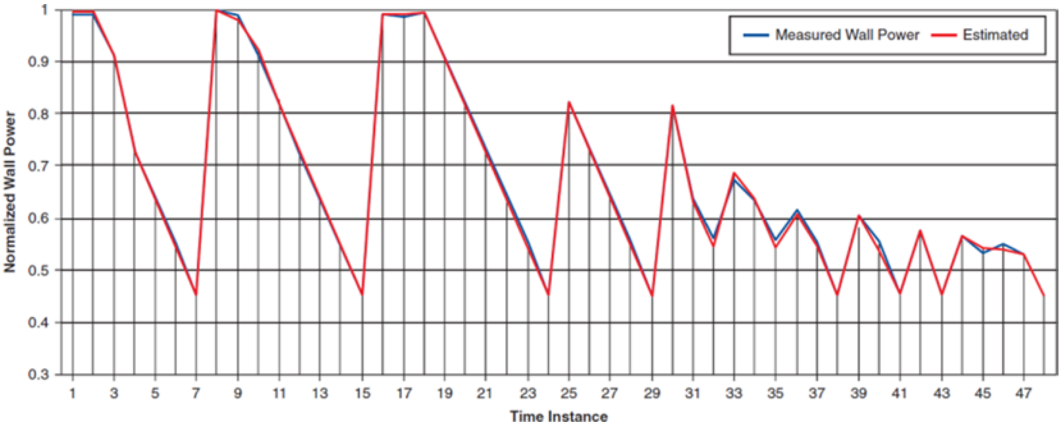


Figure 11-11. *Wall power, measured versus estimated (as function of component power)*

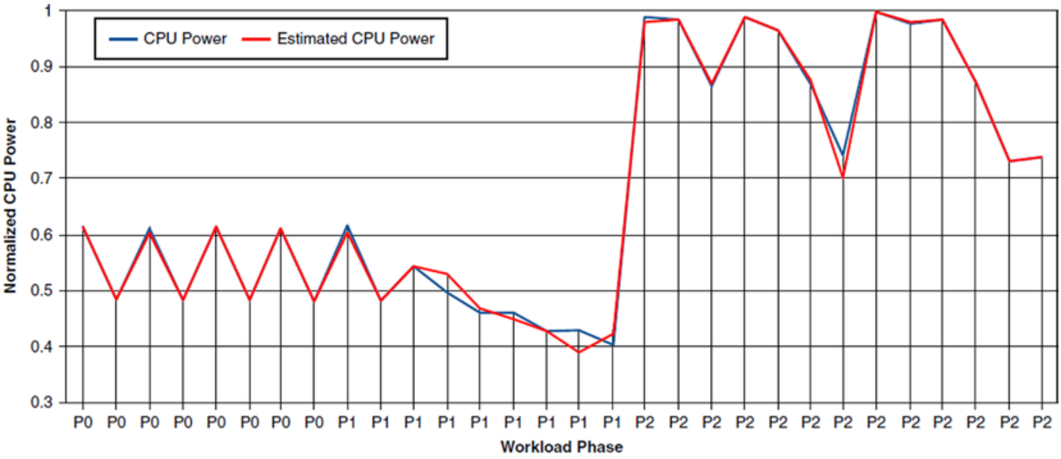


Figure 11-12. *CPU power, measured versus estimated; estimated CPU power is phase wise and based on the throughput and target latency requirements*

On average a machine learning regression function supported by SVM delivers accuracy between 97 percent and 98.5 percent. Each phase is trained for its own performance and latency model coefficients.

Figure 11-13 depicts an example consisting of four possible workload conditions on a server. On the x axis, *tt*00, *tt*10, and *tt*20 stand for think times of 0.0ms, 10.0ms, and 20.0ms, respectively. The y axis shows response times. The red multisegment line in the figure connects four workload points (W_1, W_2, W_3, W_4).

These points are randomly selected perturbations in supply and demand ; for example, W_1 results from setting a think time of 20.0ms and a CPU RAPL value of 40W; W_2 results from a think time of 0.0ms (driving a higher arrival rate than W_1) and a CPU RAPL value of 50W, and so on.

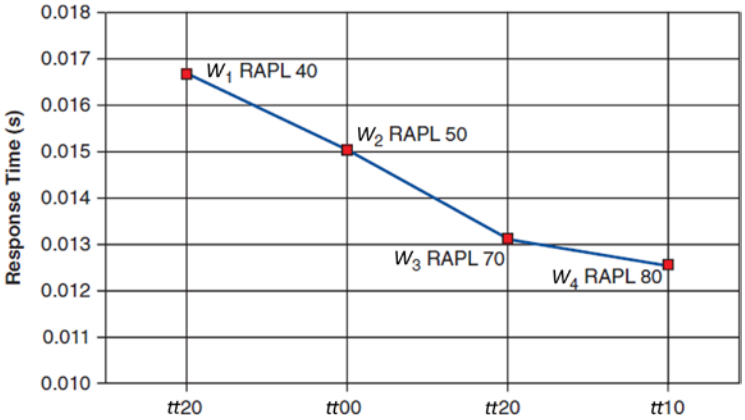


Figure 11-13. *Response time at four arbitrarily selected points, reflecting four possible workload and server conditions*

If none of the response times for W_1, W_2, W_3, and W_4 were to exceed a desired performance objective—for instance, an SLA target of $R = 20.0$ms—then it would be desirable to save power by reducing performance, so long as the higher response times were still below the target of 20.0ms. However, if at any of these workload points the response time were to exceed a desired threshold, then it would be preferable to improve performance by increasing the power to meet the SLA.

Generally, an SLA may spell out throughput and response time expectations and may include details, such as the fraction of workload that must be completed within a threshold amount of response time under differing levels of throughput. For ease of description, this example has a simple SLA setting: that the response time, averaged over small time intervals (1s), not exceed a static target value of 14.0ms; this is displayed in Figure 11-14 by the solid line, $R = 0.014$.

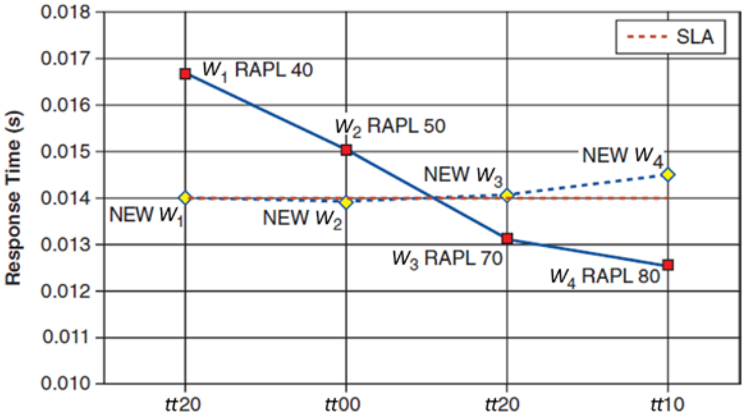


Figure 11-14. *Illustration of improvement in response time, using proactive control of CPU power employing CPU RAPL*

As you can see, new workload points (shown in diamonds) result from proactive power performance control through the use of a trained SVM model. Additionally, new RAPL settings (higher CPU power) computed using the trained model reduce the response times for W_1 and W_2 from their previous values (by 15 percent and 7 percent, respectively) to new values that are much closer to the SLA. Similarly, the model training produces lower CPU power settings for W_3 and W_4, which leads to power savings at the cost of higher response times and to 11.5 percent improvement in energy efficiency. Incidentally, the new setting for W_4 misses the SLA target by a small but not negligible margin, which could force a recomputation of the CPU RAPL setting in the next iteration. Note that to reduce frequent course correction, a control policy may permit overshooting the SLA target by a small margin in either direction. Here, because the new RAPL settings for W_1 and W_2 reduced response times, phase-aware CPU power scaling yields significant power reduction at all performance levels, relative to isolated tuning.

Example 3: System Approach to Intrusion Detection

In an era of cooperating ad hoc networks and pervasive wireless connectivity, we are becoming more vulnerable to malicious attacks. These sophisticated attacks operate under the threshold boundaries during an intrusion attempt and can only be identified by profiling the complete system activity, in relation to a normal behavior. Many of these attacks are silent in nature and cannot be detected by conventional *intrusion detection system* (IDS) methods, such as traffic monitoring, port scanning, or protocol violation. Intrusion detection may be compared to the human immune system, which, through understanding of the specifications of normal processes, identifies and eliminates anomalies. Identifiers should be distributed throughout a system with identifiable and adaptable relationships. We therefore need a model that, in each state, has a probability of producing observable system outputs and a separate probability indicating the next states.

Unlike wired networks, ad hoc nodes coordinate among member nodes to allow exclusive use of the communication channel. A malicious node can exploit this distributed and complex decision-making property of cooperating nodes to launch an attack on, or hijack, the node. This inherent vulnerability can disable the whole network cluster and further compromise security through impersonating, message contamination, passive listening, or acting as a malicious router. An IDS mechanism should be able to detect intrusion by monitoring unusual activities in the system via comparison with a user's profile and with evolving trends. Although they may not be sufficient to prevent malicious attacks if the attacker operates below the threshold, threshold-based mechanisms can be modified to monitor trends in the related system components to predict an attack. This is similar to an HMM (see Chapter 5), in which the hidden state (attack) can be predicted from relevant observations (changes in system parameters, fault frequency, and so on). Observed behavior acts as a signature or description of normal or abnormal activity and is characterized in terms of a statistical metric and model. A *metric* is a random variable representing a quantitative measure accumulated over a period of time. Observations obtained from the audit records, when used together with a statistical model, analyze any deviation from a standard profile and trigger a possible intrusion state.

This example discusses an HMM-based strategy for intrusion detection, using a multivariate Gaussian model for observations that are in turn used to predict an attack that exists in the form of a hidden state. The model comprises a self-organizing network for event clustering, an observation classifier, a drift detector, a profile estimator (PE), a *Gaussian mixture model* (GMM) accelerator, and an HMM engine. This method is designed to predict intrusion states, based on observed deviation from normal profiles or by classifying these deviations into an appropriate attack profile. An HMM is a stochastic model of discrete events and a variation of the Markov chain. Like a conventional Markov chain, an HMM consists of a set of discrete states and a matrix $A = \{a_{ij}\}$ of *state transition probabilities*. Additionally, every state has a vector of *observed symbol probabilities*, $B = b_j(v)$, which corresponds to the probability that the system will produce a symbol of type v when it is in state j. The states of the HMM can only be inferred from the observed symbols—hence, the term *hidden*. HMM correlates observations with hidden states that factor in the system design, in which observation points are optimized, using an acceptable set of system-wide *intrusion checkpoints* (ICs); hidden states are created using explicit knowledge of probabilistic relationships with these observations. These relationships (also called *profiles*) are hardened and evolve with the constant usage of the multiple and independent systems. If observation points can be standardized, then the problem of intrusion predictability can be reduced to profiling the existing and new, hidden states to standard observations.

Modeling Scheme

Parameters for HMM modeling schemes consist of *observed states*, *hidden* (*intrusion*) *states*, and *HMM profiles*. HMM training, using initial data and continuous reestimation, creates a profile that involves transition probabilities and observation symbol probabilities. HMM modeling involves the following tasks:

- Measuring the *observed states*, which are analytically or logically derived from the intrusion indicators. These indicators are test points spread throughout the system.
- Estimating the *instantaneous observation* probability function, which indicates the probability of an observation, given a hidden state. This density function can be estimated using an explicit parametric model (multivariate Gaussian) or, implicitly, from data via nonparametric methods (multivariate kernel density emission).
- Estimating the *hidden states* by clustering the homogeneous behavior of single or multiple components. These states are indicative of various intrusion activities that need to be identified to the administrator.
- Estimating the *hidden state transition* probability matrix, using prior *knowledge or random data. Prior knowledge, along with long-term temporal characteristics, indicates an approximate probability of the transitioning of state components from one intrusion state to another.

Observed (Emission) States

Observed states represent competing risks derived analytically or logically, using IC indicators. Machine intrusion can be considered a result of several components' competing for the occurrences of the intrusion. In this model the IC engine derives continued multivariate observations, which is similar to the mean and standard deviation model, except that the former is based on correlations between two or more metrics. These observations $b_j(v)$ have a continuous probability density function (PDF) and are a mixture of multivariate Gaussian (normal) distributions, expressed (Lee, Kawahara, and Shikano 2001) as

$$b_j(v) = \sum_{k=1}^{M} c_{jk}\left[\frac{1}{(2\pi)^{M/2}\,|\sigma_{jk}|^{1/2}}\,exp\left[-\frac{1}{2}(v-\mu_{jk})^T\sigma_{jk}^{-1}(v-\mu_{jk})\right]\right], \qquad (11\text{-}6)$$

where $(\cdot)^T$ denotes transpose, and

$$c_{jk} \geq 0 \;\&\; \sum_{k=1}^{M} c_{jk} = 1$$

σ_{jk}= covariance matrix of the kth mixture component of the jth state

μ_{jk} = mean vector of the kth mixture component of the jth state

v = observation vector

M = number of dimensions of an observation with a multivariate Gaussian distribution

$\theta_{jk} = (\sigma_{jk}, \mu_{jk})$ = Gaussian components

η_{jk} = drift factor of the kth mixture component of the jth state

$\lambda_{jk} = (\theta_{jk}, c_{jk}, \eta_{jk})$ = user profile components

It is the responsibility of the IC engine to reestimate the λ_{jk} parameters dynamically for all matrices and all possible attack states. Various matrices that represent dimensions of an observation are as follows:

- *Resource activity trend*: The measure of a resource activity that is monitored over a larger sampling period and that has characteristics that repeat over that sampling period. Each period of activity can be thought of as an extra dimension of activity measure.
- *Event interval*: The measure of an interval between two successive activities (e.g., logging attempts).
- *Event trend*: The measure of events monitored over a larger sampling period, with the objective of calculating the event behavior with a built-in repeatability (e.g., the count of logging attempts in a day).

Hidden States

Hidden states $\boldsymbol{S} = \{S_1, S_2, \cdots, S_{N-1}, S_N\}$ are a set of states that are not visible, but each state randomly generates a mixture of the *M* observations (or visible states *O*). The probability of the subsequent state depends only on the previous state. The complete model is defined by the following probabilities: *transition probability matrix* $\boldsymbol{A} = a_{ij}$, where $a_{ij} = p(S_i|S_j)$; *observation probability matrix* $\boldsymbol{B} = (b_i(v_m))$, where $b_i(v_m) = p(v_m|S_i)$; and an initial probability vector $\boldsymbol{\pi} = p(S_i)$. *Observation probability* represents an attribute that is observed with some probability if a particular failure state is anticipated. The model is represented by $\boldsymbol{M} = (\boldsymbol{A},\boldsymbol{B},\boldsymbol{\pi})$.
A *transition probability matrix* is a square matrix of size equal to the number of states and stands for the state transition probabilities.

The observation probability distribution is a nonsquare matrix whose dimensions equal the number of states by number of observables. This distribution represents the probability of an observation for a given state. The IDS depicted in Figure 11-15 uses these states:

- *Normal* (N) state indicates profile compliance.
- *Hostile intrusion attempt* (*HI*) indicates a hostile intrusion attempt that is in progress. This is typical of an external agent trying to bypass the system security.
- *Friendly intrusion attempt* (*FI*) denotes a nonhostile intrusion attempt that is in progress. This is typical of an internal agent trying to bypass the system security.
- *Intrusion in progress* (*IP*) signals an intrusion activity that is setting itself up. This includes attempts to access privileged resources and acceleration in resource usage.
- *Intrusion successful* (*IS*) signifies a successful intrusion. Successful intrusion will be accompanied by unusual resource usage (CPU, memory, I/O activity, and so on).

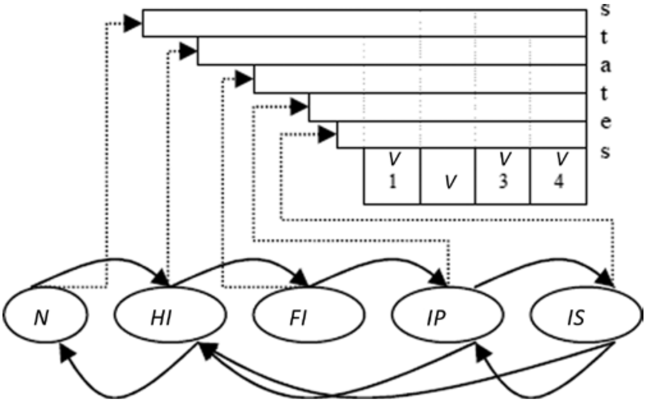


Figure 11-15. *HMM model, with five intrusion states and four Gaussian distributions for each state; each Gaussian distribution can be represented as the mixture component of an observation. (Source: Khanna and Liu 2006)*

Intrusion Detection System Architecture

In ad hoc networks an IDS is deployed at the nodes to detect the signs of intrusion locally and independently of other nodes, instead of using routers, gateways, or firewalls. In this section, we define components of the IDS that cooperate with each other to predict an attack state.

After the model is trained, it enters a runtime state, in which it examines and classifies each valid observation. The model then decides to add the observation to a profile update, reject it, or mark it "unclassified." This decision is important, because a drift in the user's normal behavior may represent an attack situation. An unclassified observation is monitored for classification in the future. This observation will later be rejected as a noise or classified as a valid state, based on the trending similarity between unclassified states tending toward a certain classification and on feedback from the state machine resulting from other, independent observations. Various components of an IDS are as follows:

> **Profile estimator** (**PE**): The PE is responsible for maintaining/reestimating user profiles, classifying an observation as an attack state, triggering an alert upon detecting a suspicious observation, or acting on the HMM feedback for reestimation of a profile. User profile data consist of PDF parameters $\lambda_{jk} = (\sigma_{jk}, \mu_{jk}, c_{jk}, \eta_{jk})$, where *j* represents the intrusion state, and *k* stands for the GMM mixture component. A new observation is evaluated against this profile, which results in its classification and drift detection.
>
> **Instrumentation**: Instrumentation produces event data, which are processed and used by a *clustering agent* to estimate the profile. Component identification and measurements involve setup to discern whether events should be sampled at regular intervals or whether notification (or an alert) should be generated as an event vector upon recording changes in pattern. The sensor data should be able to analyze data, either as they are collected or afterward, and to provide real-time alert notification for suspected intrusive behavior. This will require fast-acting silicon hooks that are capable of identifying, counting, thresholding, timestamping, eventing, and clearing an activity. Examples of such hooks are performance counters, flip counters (also called transaction counters), header sniffers, fault alerts (page faults, and so on), and bandwidth usage monitors.

At the same time, software instrumentation is also required to sample software-related measurements, such as session activity, system call usage between various processes and applications, file system usage, and swap-in/swap-out usage. Most operating systems support these hooks in the form of process tracking (such as process ID [PID], in UNIX). Combinations of these fast-acting hooks with sampling capability are clustered to enact an observation.

Data clustering: Observation data are dependent on the aggregation of events that are active. For instance, a resource fault event generated by a resource usage engine is further categorized as a *fault type*, such as a page fault. Page faults count, and *invalid page faults* in a sampled interval represent instances of measurement (m_1,m_2). An observation (emission) can be a set of correlated measurements but is represented by a single probability distribution function. Each of these measurements carries different weights, as in multivariate Gaussian distribution. For example, disk I/O usage may be related to network I/O usage because of the network file system (NFS). Such a relationship is incorporated into the profile, for the completeness of the observation, and reduces the dimensionality, for effective runtime handling.

Classifier: Observation data are analyzed for the purpose of subclassification as an appropriate attack state in a profile driven by different probability distribution parameters. Once the appropriate attack state is identified, an attention event is generated to initiate a corrective or logging action. Observations are also analyzed for concept drift to compensate for changes in user (or attack) behavior. Therefore, one of the objectives of the IC engine (see Figure 11-16) is to build a classifier for j (attack states) that has a posterior probability $p(j|v)$ close to unity for one value of j and close to 0 for all the others, for each realization. This can be obtained by minimizing the Shannon entropy, given observed data v, which can be evaluated for each observation as

$$E = -\sum_{j=1}^{M} p(j\,|\,v)\log(p(j\,|\,v)). \tag{11-7}$$

Each IC engine samples its observations independently of other observations (or emissions). Whenever it suspects an abnormal activity, it triggers an alert, which causes an evaluation of the most likely state. As the system changes its active behavior, the profile corresponding to that behavior is updated to avoid false-positive evaluations by reevaluating the model parameters, using continuous estimation mechanisms in real time. New HMM parameters are evaluated again against the historical HMM parameters by comparing the entropy of the old and the retrained models. The *expectation maximization* (EM) algorithm (Moon 1996) provides a general approach to the problem of *maximum likelihood estimation* (MLE) of parameters in statistical models with variables that are not observed. The evaluation process yields a parameter set, which the algorithm uses to assign observation points to new states. The computational complexity of the EM algorithm for GMMs is $O(i \times ND^2)$, where i is the number of iterations performed, N is the number of samples, and D is the state dimensionality. A common implementation choice is the k-means algorithm, in which k clusters are parameterized by their centroids, with a complexity of $O(kND)$. A number of other algorithms can also be used, including x-means clustering (Pelleg and Moore 2000), which reduces the complexity to $O(D)$.

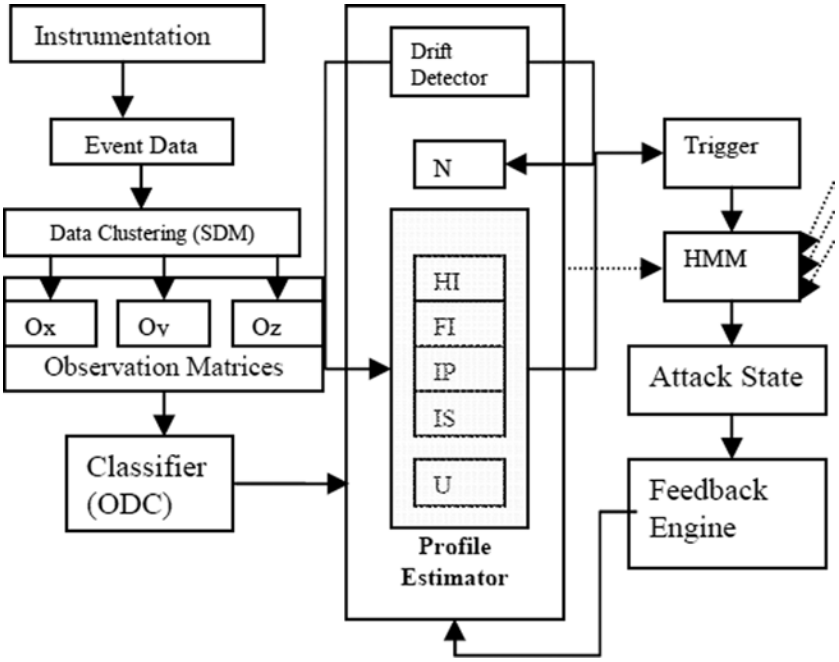


Figure 11-16. *IC engine. Reestimation of the profile uses an observation classifier and HMM feedback to the profile. The profile manager triggers an attention event if the observation classifies as an attack state or cannot be classified (U). An attention event initiates an HMM state sequence prediction, based on other, continuous observations (dotted arrows), extracted in conjunction with profiles and state transition probabilities. (Source: Khanna and Liu 2006)*

Concept drift detector (**CDD**): This module detects and analyzes the concept drifting (Widmer and Kubat 1996) in the profile, when the training dataset alone is not sufficient, and the model (profile) needs to be updated continually. When there is a time-evolving concept drift, using old data unselectively helps if the new concept and old concept still have consistencies and if the amount of old data chosen arbitrarily happens to be right (Fan 2004). This requires an efficient approach to data mining that aids in selecting a combination of new and old data (historical) to make an accurate reprofiling and further classification. The mechanism used is the *Kullback-Leibler* (KL) *divergence* (Kullback and Leibler 1951), in which relative entropy measures the kernel distance between two probability distributions of generative models. The KL divergence is also the gain in Shannon information that occurs in going from the a priori to the posteriori, expressed as

$$\alpha_{jkt} = KL(b_j(v \mid \theta'_{jkt}), b_j(v \mid \theta_{jkt})), \tag{11-8}$$

where α_{jkt} is the KL divergence measure, θ'_{jkt} is the new Gaussian component, and θ_{jkt} is the old Gaussian component of the *k*th mixture of the *j*th state at time *t*. You can evaluate divergence via a Monte Carlo simulation, using the law of large numbers (Grimmett and Stirzacker 1992), which draws an observation v_i from the estimated Gaussian component θ'_{jkt}, computes the log ratio, and averages this over *M* samples as

$$\alpha_{jk} \approx \frac{1}{M}\sum_{i=1}^{M}\log\left(\frac{b_j(v_i \mid \theta'_{jkt})}{b_j(v_i \mid \theta_{jkt})}\right). \tag{11-9}$$

KL divergence data calculated in the temporal domain are used to evaluate the speed of the drift (also called the *drift factor*) ($0 \leq \eta \leq 1$). These data are then used to assign weights to the historical parameters, which are in turn used for reprofiling.

Feedback engine (**FE**): This component is responsible for feeding back the current state information to the PE. The current state information is reevaluated, using the current PDF model parameters. This reevaluated state information is then used for improving the descent algorithm for finding the MLE.

Profiles and System Considerations

In this section, we look at events that form input to the profile structure. We define the features as processed observations derived from one or more temporal input events, using a processor function.

Exploiting temporal sequence information of events leads to better performance (Ghosh, Schwartzbard, and Schata 1999) of the profiles that are defined for individual users, programs, or classes. Abnormal activity in any of the following forms is an indicator of an intrusion or a worm activity:

- *CPU activity* is monitored by sampling faults, interprocessor interrupt (IPI) calls, context switches, thread migrations, spins on locks, and usage statistics.
- *Network activity* is monitored by sampling input error rate, collision rate, remote procedure call (RPC) rejection rate, duplicate acknowledgment (DUPACK), retransmission rate, time-out rate, refreshed authentications, bandwidth usage, active connections, connection establishment failure, header errors and checksum failures, and so on.
- *Interrupt activity* is monitored by sampling device interrupts (nontimer).
- *I/O utilization* is monitored by sampling the I/O requests' average queue lengths and busy percentages.
- *Memory activity* is monitored by sampling memory transfer rate, page statistics (reclaim rate, swap-in rate, swap-out rate), address translation faults, and pages scanned and paging averages over a short interval.
- *File access activity* is monitored by sampling file access frequency, file usage overflow, and file access faults.
- *System process activity* is monitored by sampling processes with inappropriate process priorities, CPU and memory resources used by processes, processes' length, processes that are blocking I/Os, zombie processes, and the command and terminal that generated the process.

- *System fault activity* represents an illegal activity (or a hardware error) and is sampled to detect abnormality in the system usage. Rare faults are a result of bad programming, but spurts of activity indicate an attack.
- *System call activity* involves powerful tools for obtaining computer system privileges. An intrusion is accompanied by the execution of unexpected system calls. If the system call execution pattern of a program can be collected before it is executed and is used for comparison with the runtime system call execution behavior, then unexpected execution of system calls can be detected. During real-time operation a pattern-matching algorithm is applied to match on the fly the system calls generated by the process examined with entries from the pattern table. Based on how well the matching can be done, it is decided whether the sequence of system calls represents normal or anomalous behavior (Wespi, Dacier, and Debar 1999).
- *Session activity* is monitored by sampling logging frequency, unsuccessful logging attempts, session durations, session time, and session resource usages.

Sensor Data Measurements

Sensor data are collected and statistically processed so that they can be used to measure historical trends, capture unique patterns, and visualize abnormal behavior. The data are classified and then analyzed for use in prediction of abnormal activity. Sensor data measurements comprise various components that perform either a statistical processing function or an infrastructure function (such as generating priority events), as follows:

> **Sensor data measurement** (**SDM**) hooks reduce system complexity and increase the possibility of software reuse (see Figure 11-17). SDM accelerates the combined measurement of the clustered components with an ability to send alerts, using a system's policy. Hardware and software act as glue between transducers and a control program that is capable of measuring the event interval and the event trend and of generating alerts upon deviation from normal behavior, as defined by system policy. The SDM hardware exists as a multiple-instance entity that receives alert vectors from various events spread throughout the system. A set of correlated events that forms a cluster is registered against a common SDM instance. This instance represents the Bayes optimal decision boundaries between a set of pattern classes, with each class represented by an SDM instance and associated with a reference vector. Each SDM instance can trend and alert and integrates the measurements from the event sensors into a unified view. Cluster trending analysis is very sensitive to small signal variations and capable of detecting the abnormal signals embedded in the normal signals via supervised self-organizing maps (Kohonen 1995), using *learning vector quantization* (LVQ). The strategy behind LVQ is to effectively train the reference vectors to define the Bayes optimal decision boundaries between the SDM classes registered to an SDM instance.

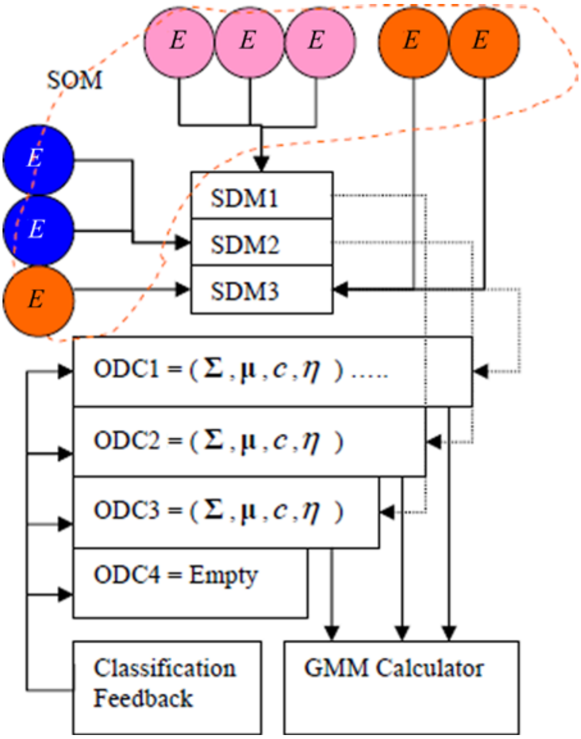


Figure 11-17. *Illustration of the relationship between events (circles), sensors (SDM), and classifiers (ODC). Clusters of events (marked by common colors) are registered to an SDM. Upon evaluating the event properties, the SDM generates an event to ODC, which is responsible for classification, trend analysis, and drift calculation. Classification feedback acts as a mechanism for reestimation. (Source: Khanna and Liu 2006)*

Observation data classifier (**ODC**) hooks accelerate the classification of an observation alert generated by SDM. This is multiple-instance hardware capable of handling multiple observations in parallel. Each registered observation instance of the ODC hook consists of Gaussian probability distribution parameters for each state. Upon receiving an SDM alert, the corresponding observation is then classified as a specific state. Reclassification of observed data may cause changes in the probability distribution parameters corresponding to the state. ODC can maintain the historical parameters, which are used to calculate concept drift properties, such as drift factor and drift speed, using the KL drift detector.

The **GMM calculator** calculates the probability of the Gaussian mixture for each state, using the current observation. During system setup, event vectors are registered against SDM instances. These events are clustered and processed in their individual SDMs. The processing includes trigger properties, which initiate an observation. These observations then act as single-dimensional events that are registered to their ODC. Upon receiving the trigger, ODC performs reclassification of the observation derived from the trigger and calculates the concept drift. This hardware is activated upon a trigger by its parent.

Summary

As more and more data are expressed digitally in an unstructured form, new computing models are being developed to process that data in a meaningful manner. Machine learning methods can be applied to synthesize the fundamental relationship between the unstructured datasets and information through systematic application of algorithms. Machine learning exploits the power of generalization that is an inherent and essential component of concept formation through human learning. The machine learning methodology can be applied to develop autonomous systems, using modular functions to enact an intelligent feedback control system. This approach can play a critical role in modeling the knowledge function, which is used to enact a stable and viable system. This chapter presented three examples of techniques used in machine learning. The first example employed the concept of workload fingerprinting, using phase detection to establish observable characteristics exhibiting spatial uniformity and distinctiveness. The second example was based on the concept of optimal, dynamic energy distribution among multiple compute elements. This example proposed phases as compressed representative output that can be used in conjunction with any other statistical parameter to predict future behavior. The last example suggested use of the IDS mechanism for detecting intrusions by monitoring unusual activities in the system with reference to the user's profile and evolving trends. Each example used an application-specific grouping of machine learning techniques to achieve the desired goals.

References

Becker, Suzanna, and Geoffrey E. Hinton. "Self-Organizing Neural Network that Discovers Surfaces in Random-Dot Stereograms." *Nature* 355, no. 6356 (1992): 161–163.

Boser, Bernhard E., Isabelle M. Guyon, and Vladimir N. Vapnik. "A Training Algorithm for Optimal Margin Classifiers." In *COLT '92: Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, 144–152. New York: ACM, 1992.

Fan, Wei. "Systematic Data Selection to Mine Concept-Drifting Data Streams." In *KDD '04: Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 128–137. New York: ACM, 2004.

Ghosh, Anup K., Aaron Schwartzbard, and Michael Schatz. "Learning Program Behavior Profiles for Intrusion Detection." In *ID' 99: Proceedings of the 1st Conference on Intrusion Detection and Network Monitoring*. Berkley, CA: USENIX, 1999.

Grimmett, Geoffrey, and David Stirzaker. *Probability and Random Processes*. Oxford: Clarendon, 1992.

Khanna, Rahul, and Huaping Liu. "System Approach to Intrusion Detection Using Hidden Markov Model." In *Proceedings of the 2006 International Conference on Wireless Communications and Mobile Computing*, 349–354. New York: ACM, 2006.

Kohonen, Teuvo. "*Self-Organizing Maps, Third Edition*." Berlin: Springer, 1995.

Kullback, Solomon, and Richard A. Leibler. "On Information and Sufficiency." *Annals of Mathematical Statistics* 22, no. 1 (1951): 79–86.

Lee, Akinobu, Tatsuya Kawahara, and Kiyohiro Shikano. "Gaussian Mixture Selection Using Context-Independent HMM." In *Proceedings of the 2001 IEEE International Conference on Acoustics, Speech, and Signal Processing,* 69–72. Piscataway, NJ: Institute of Electrical and Electronics Engineers, 2001.

Moon, Todd K. "The Expectation-Maximization Algorithm." *IEEE Signal Processing Magazine* 13, no. 6 (1996): 47–60.

Pelleg, Dan, and Andrew W. Moore. “*X*-Means: Extending *K*-Means with Efficient Estimation of the Number of Clusters.” In *ICML ’00: Proceedings of the Seventeenth International Conference on Machine Learning*, 727–734. San Francisco: Morgan Kaufmann, 2000.

Siddha, Suresh. “Multi-Core and Linux Kernel.” Technical report, *Intel Open Source Technology Center*, 2007.

Wespi, Andreas, Marc Dacier, and Hervé Debar. “An Intrusion-Detection System Based on the Teiresias Pattern-Discovery Algorithm.” In *EICAR Proceedings 1999*, edited by Urs E. Gattiker, Pia Pedersen, and Karsten Petersen, 1–15. Aalborg, Denmark: Tim-World, 1999.

Widmer, Gerhard, and Miroslav Kubat. “Learning in the Presence of Concept Drift and Hidden Contexts.” *Machine Learning* 23, no. 1 (1996): 69–101.