

CARVE: A Cognitive Agent for Resource Value Estimation

Jonathan Wildstrom*, Peter Stone, Emmett Witchel
Department of Computer Sciences
The University of Texas at Austin
{jwildstr,pstone,witchel}@cs.utexas.edu

Abstract

Recently, industry has begun investigating and moving towards utility computing, where computational resources (processing, memory and I/O) are available on demand at a market cost. On-demand access to computational resources enables fine-grained resource allocation for web-based applications, e.g., the possibility of provisioning for a minimum workload while allowing the rental of additional resources for unexpected workload changes. However, renting additional resources relies on the ability to quickly and accurately estimate the value of the resource. This paper introduces CARVE: a Cognitive Agent for Resource Value Estimation. CARVE is a machine-learning based approach that learns to predict the change in system value of having more or less system resources. Using only low-level statistics and with no custom instrumentation of the operating system or middleware, CARVE is able to make informed decisions about the return on investment of physical memory when implemented and evaluated on a partitioned system running a multi-partition, multi-process distributed benchmark. We show that CARVE is competitive with static choices of computing resources over a variety of test workloads and also has the ability to outperform all static configurations.

1. Introduction

Recent years have seen a growing push toward *utility computing* [13, 6]. In this new paradigm, computation and memory are treated as a utility similar to electricity or water, and “service capacity is provided as needed and the customer pays only for actual use.” [3]. In a utility computing system, decisions about provisioning machines become far more flexible. Instead of needing to decide on a single hardware configuration that would suffice in all situations, administrators can now exercise finer-grained con-

trol of the specific resource allocation for a system as dictated by the workload. This allows a system to be initially provisioned for the expected workload, with the ability to expand the system for periods of increased activity (e.g., the holiday season for retailers). Data centers can save on power and cooling during times of lesser activity. While an administrator would normally use a utility computing model to avoid hosting any of their system, virtualization and hardware partitioning allow the administrator to retain more control of their system, while achieving a similar goal. In this scenario, the business would consider the savings on power and cooling of the systems as being the gain in a lesser-allocation scenario.

Reprovisioning of hardware resources can be done in different ways. One possibility would be overprovisioning the physical system as a whole, but holding some resources “in reserve.” Reserve resources either save power or are given to a different virtualized system. This is a limited form of overprovisioning, with a single pool of excess resources for the business as a whole.

Another option is renting the resources from a separate “owner.” One example of this would be IBM’s Capacity on Demand [8], in which some resources can be activated for a period of time at a cost, while a base set of resources is owned and always available. Another possible source of resource control would be through a utility computing infrastructure, such as Amazon’s Elastic Compute Cloud [1]. Such an infrastructure could impose a pricing model for resources such as memory and processing power, in which a customer can increase or decrease their resources at will.

One important question which needs to be addressed in any of these provisioning scenarios is when additional resources are a worthwhile investment. To determine the answer to this question, an administrator must be able to predict the value of those additional resources and compare this value to the expense of activating the resources. For example, increasing the resource allocation to a system may allow an online store to return product information pages in 200 ms instead of 2 seconds; this increased response time has a value in terms of customer satisfaction and additional

*currently employed by IBM Systems and Storage Group. Any opinions expressed in this paper may not necessarily be the opinions of IBM.

orders. This additional value, however, may not exceed the cost of the additional resources.

This work introduces CARVE: a Cognitive Agent for Resource Value Estimation. CARVE is an autonomous agent that learns to estimate the value of additional memory resources and make decisions aimed at ensuring a positive return on investment. CARVE is able to compete with static choices of provisioning over a variety of test workloads and resource costs. Although this work is focused on physical overprovisioning, the results should be applicable to all three motivating scenarios.

This paper is organized as follows. The next section gives an overview of the hardware and software used in our testbed. Section 3 describes the training of CARVE’s learned model, as well as its implementation and evaluation. Section 4 details the results of our experiments and contains some discussion of their implications. Section 5 gives an overview of related work, and Section 6 concludes.

2. Experimental Testbed and System Overview

To set up an interesting testbed for this work, we choose to investigate a multi-tier Internet application. By using a multi-tier application, we can use partitioning and virtual networking to separate the pieces of the application into logically independent units, providing isolation from resource contention and allowing targeted resource decisions. The ability to have multiple independent logical partitions on a single piece of hardware has only recently become widely available. Although previous work [22, 20] has simulated this single partitionable hardware on separate physical machines, this work implements the entire application on only one physical machine, which is more reflective of the intended use of our contribution.

2.1. Hardware

Although the two-tier application runs entirely on one physical system, our overall experimental testbed includes 3 separate machines. The heart of the testbed is a partitionable IBM System p5 55A with 4 1.65 GHz processors and 16 GB of memory. The system is running three shared processor partitions: one for each of the tiers of the application, and a Virtual I/O Server (VIOS). The VIOS abstracts the disks and network for the other two partitions, and runs VIOS Fixpack 9.2, using 0.5 processing units¹, running one virtual processor, and 1GB of memory. Each of the application partitions has a baseline setting of 1 processing unit, running 2 virtual processors, and 2 GB of memory. These partitions are running AIX 5.3, Technology Level 6. Additionally, a kernel extension was developed and installed to latch into the dynamic reconfiguration framework and

¹A processing unit represents one physical CPU.

record information about each resource change; this data is used to calculate the ongoing cost of resources. Finally, AIX’s Workload Manager is configured to place the benchmark processes in a lower tier than the system services; this allows the network connections associated with reconfiguration events to take priority over the benchmark and not delay the resource reallocation.

The partitionable system is managed by a Hardware Management Console (HMC), which initiates all resource reallocation events. This system is a customized x86 machine running version 6.1.2 of the HMC software. Finally, the application is tested externally from a IBM ThinkCentre M50 with a 3 GHz processor and 2 GB of memory. In addition to testing the application, this system also hosts CARVE².

2.2. Workload Generation and Value Function

In order to have some level of confidence that an automatic investment analysis can work on a real system, it is important to simulate a realistic workload. For this work, we use the TPC-W benchmark [16]. This is a standardized benchmark which simulates an online bookstore, with specified expected response times³. A full description of the benchmark can be found in the specification, and summaries and analyses are also available [7, 14]. A full discussion of modifications made to the benchmark is in a previous work by the authors [21]. An abstraction of the TPC-W implementation on our systems can be seen in Figure 1.

A typical run of the benchmark consists of a given workload being run against the System Under Test (SUT). This workload consists of Emulated Browsers (EBs), which request a series of web pages chosen from a pool of 14 dynamically generated pages. These pages are divided into 6 *browsing* pages, which only read from the database, and 8 *ordering* pages, which can perform updates as well. The relative probabilities of page accesses are defined based on which of three *mixes* an EB is running. These are:

1. *browsing* mix: This mix is primarily reads of the database, with 95% of accesses being browsing pages.
2. *shopping* mix: This is the “normal” state of the benchmark, with 80% of accesses being browsing pages and 20% being ordering pages.
3. *ordering* mix: In this mix, the database is under the most stress, with 50% of accesses being to ordering pages.

Normally, one mix is run by all EBs; however, to allow more scenarios in our work, we allow all three mixes to be

²While it would be optimal to host CARVE on the HMC, the closed design of the system makes this implausible.

³TPC-W was recently extended and superseded by TPC-App.

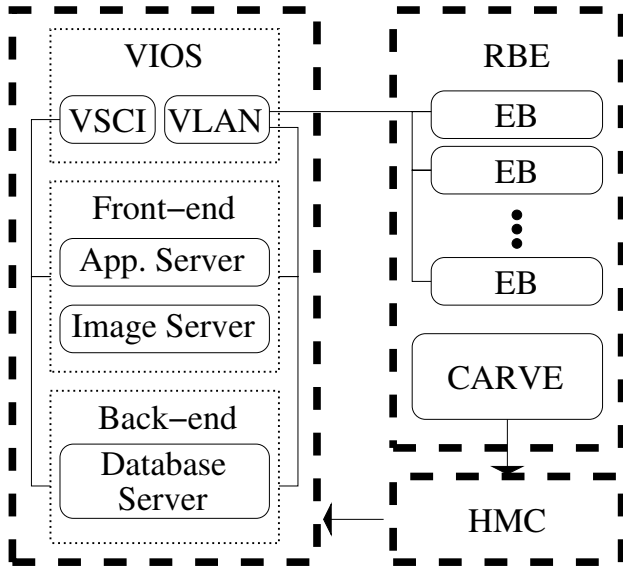


Figure 1. The machines used in the physical setup. The thick, dashed rectangles represent physical machines; the dotted rectangles are partitions, and the rounded rectangles are logical units. Network connections are shown as lines and come together at the point where they are managed.

run simultaneously by different EBs. A single run of the benchmark consists of a warm-up period, during which the applications are brought into memory and common data is cached, and then a Measurement Interval (MI) during which the performance of the system is monitored. Finally, there is an optional ramp-down period. For our work, we always use a 240 second ramp-up period and a 100 second ramp-down period; the measurement intervals vary with the specific run and are detailed later.

Because the TPC-W specification merely indicates expected response times and a satisfaction level for compliance, we needed to infer a Service Level Agreement (SLA) that could describe the value of a specific transaction. For this work, we use a step function. A certain value is given to transactions that satisfy the response time requirement, which we denote by 1 unit of value. Transactions which violate the threshold incur a penalty that begins at 1 and increases in 4 stages to a maximum value of 10. This maximum is chosen to correspond to the 90% compliance limit imposed by the specification. This maximum value is attained at 1.5 times the expected response time. A graphical representation of this SLA can be seen in Figure 2.

2.3. Software

At its simplest, TPC-W needs three pieces of software: the application server, an image server, and a database.

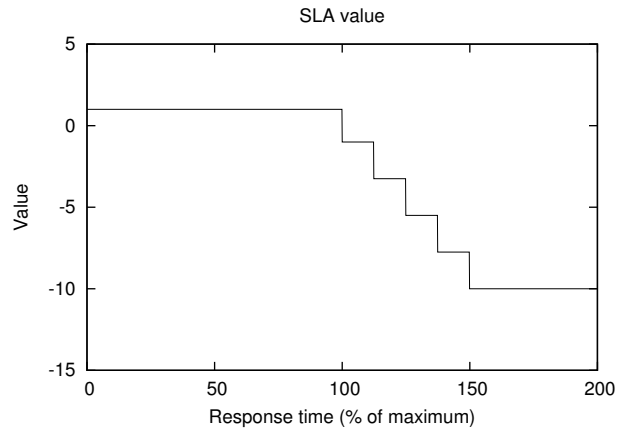


Figure 2. The value of a transaction as compared to its response time, measured as the percentage of the expected maximum response time.

Our application server is Apache Tomcat 6.0.14, which also functions as an image server. The database used in this work is PostgreSQL 8.2.4.

The application server also needs Java servlets to dynamically generate the web pages used by the benchmark. These servlets are derived from code freely available from the University of Wisconsin PHARM project [4]. This implementation includes not only the Java code for the front-end, but also an implementation of the Remote Browser Emulator (RBE) which drives the benchmark. Slight modifications were made to this code both to support Tomcat and PostgreSQL and to improve performance and stability; these modifications are explained in previous work [20].

3. CARVE: A Cognitive Agent for Resource Value Estimation

In order to autonomously perform the needed resource reallocations, CARVE needs the ability to evaluate the current status of the testbed. Given this status, CARVE must be able to predict the change in value, in terms of the defined SLA, of altering its memory configuration. From this information, CARVE can make investment analysis decisions involving a cost/benefit analysis of renting more resource (or, equivalently, giving up currently held resources). Autonomously making this decision can maximize the performance of the system, while controlling the overall price.

This prediction and decision task involves 3 stages: training CARVE's models, evaluating the on-line machine state, and performing a reallocation, if necessary. Section 3.1 describes the training and implementation of CARVE, while Section 3.2 details the testing methodology used in determining CARVE's efficacy.

3.1. Training CARVE’s model

Because we intend for CARVE to determine the return on investment for memory resources, we need learned models to predict two versions of the system value: one for the system with one unit more memory, and one for the system with one unit less memory than it currently has. In particular, we want to predict the *per-second* performance value of the system, in terms of the SLA, as we measure the cost of resource per second of usage. Finally, we would like to make this prediction using only low-level kernel statistics. These statistics are simple to collect, and are independent of middleware and application choices. We hypothesize that given examples of these low level statistics combined with low-level information about the performance of the machine, a pair of useful learned models can be inferred. One of these models will predict how performance would change if the machine were running the same workload, but with one more unit of memory, and one model will perform an equivalent prediction for one less unit of memory.

For this work, we consider 512 MB as the granularity of memory that can be added or removed at a time, although a finer granularity would also be reasonable. Although nothing in this work is directly dependent on this exact granularity (the hardware granularity is 64MB), this value was chosen to keep the static search space reasonable. This has two effects: the first is that training runs are of a tractable size, and the second is a limit of the total delay experienced in moving from the minimum to the maximum memory threshold, while accounting for hysteresis. We focus on the database partition because it is far more memory intensive than the web-server partition; experimentation with different quantities of memory on the web-server showed no performance change.

In order to train the performance prediction models, we gather training data that demonstrates the system value for a range of configurations and workloads. 100 random workloads are generated, which are used as a sampling of the possible workloads the system might experience. Each of these workloads is generated as follows: first, the total number of Emulated Browsers (EBs) to run, E_{tot} , is randomly selected to be between 800 and 1200. Two further random values are then selected, with $E_1 \in [0, E_{tot}]$ and $E_2 \in [0, E_{tot} - E_1]$. Finally, $E_3 = E_{tot} - E_1 - E_2$. The triple $\{E_1, E_2, E_3\}$ is then randomly permuted into $\{M_1, M_2, M_3\}$, and the final workload consists of M_1 browsing users, M_2 shopping users, and M_3 ordering users. The random permutation is performed to eliminate the bias towards E_1 .

Given each of these 100 workloads, we perform five runs of the benchmark with a measurement interval of 300 seconds on each. These five runs give the database partitions between 2 GB and 4 GB of memory, stepping by our gran-

ularity of 512 MB. These 500 total runs of the benchmark form the basis for our training set. Although individual runs may be abnormally good or bad (the runs can exhibit a high variance), the large number of randomly chosen workloads help mitigate this effect.

One philosophy behind our work is that it ought to be application independent. That is, it should work without any instrumentation of the software running on the system. Thus we limit ourselves to low-level system statistics as the inputs to CARVE and show that they are sufficient for making this return on investment decision. Previous work has indicated that there is potential for this approach to work for processing resources [20]; our work here extends this to memory resources. In order to use these statistics, we need to gather sample data during our training run. These statistics are sampled for 180 seconds during the measurement interval of the training runs using the *vmstat* command. We limit the interval to 180 seconds to ensure that, due to startup delays, we overlap neither of the ramp-up or ramp-down periods. Additionally, as explained below, we must use a multiple of 30 seconds.

The default version of *vmstat* is modified slightly to allow it to run as a high priority process (so it is not blocked in reporting data by the benchmark) and to report memory information as a percentage of physical memory. Because all possible statistics are represented as percentages, we do not train models specially for different given current configurations, instead allowing CARVE to learn to predict for all configurations.

The 20 statistics which *vmstat* collects can be seen in Table 1. They fall into 5 high-level categories: kernel thread counts, memory utilization, paging events, system events, and CPU usage. Some of these parameters bear additional explanation. When the virtual memory manager in AIX needs more memory, it performs a scan-and-free cycle. The “pages scanned” and “pages freed” counts track this operation. Additionally, when pages are sent to swap space, they are still counted as “active”, so the percentage of active pages can exceed 100%. Finally, the use of shared-processor partitions adds the “number of physical processors” and “percent of entitled capacity” fields. The former indicates how many physical processors are being consumed, while the latter indicates this as a percentage of the partitions entitlement (the maximum number of physical processors it may consume). These two parameters report the same information, but in different ways.

These 20 statistics form the input representation for the learned model; statistics are only collected on the database machine for this work. The current configuration is not supplied, as we would like to be able to learn a single model independent of the configuration. The 200 seconds of data are divided into 6 non-overlapping 30 second intervals, and the average value of each statistic is taken over the interval.

kernel threads (number)	runnable raw I/O blocked	blocked
VM (as % phys.)	active pages	free pages
paging (pages/sec)	file in swap pages in pages freed	file out swap pages out pages scanned
System events (number/s)	device interrupts context switches	system calls
CPU (%)	user idle # physical used	system I/O wait % entitled used

Table 1. The statistics reported by *vmstat*.

This process gives 6 input vectors. To determine the target output, we consider the difference in raw per-second SLA value between a given configuration and the configuration with 512 MB more memory. This difference is the gain we would expect to see from adding more memory. An equivalent consideration is made for 512 MB less memory to compute target values for the second learned model. The total yield is 2,400 data points for training each learned model⁴.

From these data sets, the WEKA [23] package can be used to learn a regression model to predict the gain or loss from more or less memory. The WEKA package implements many common classification and regression algorithms; for this work, we chose to use $M5'$ trees. Simple neural networks were also considered, however $M5'$ trees appeared to have a better accuracy in predicting the value.

Because $M5'$ trees combine a decision tree with linear function approximators at the leaves, they tend to be complicated and difficult to display in their entirety. One rule and function approximator for decreasing memory can be found in Figure 3. The approximator for increasing memory is much simpler, using only the amount of active memory as a decision variable and inferring 3 linear function approximators depending on what percentage of physical memory is active. One rule corresponds to the situation where the system has less than 100% active memory, and the other 2 are for between 100% and 119.7%, and over 119.7%.

3.2. Evaluating the learned model

These learned models are evaluated by testing them on newly generated random workloads. Although generated similarly to the previous training workloads, these test workloads have the distinction that they are not constant through the workload. Instead, the workloads consist of three separate 300 second phases. The workload for each phase is generated independently randomly as before. 5 such workloads are generated and used for testing; the workloads discussed in the remainder of this work are listed in Table 2.

⁴The 5 configurations allow for 4 comparisons per workload.

In order to make online investment decisions, CARVE must have information about the current state of the database partition⁵. As during the training runs, a high-priority *vmstat* runs on the database machine; however, instead of keeping a 200 second log of the statistics, they are processed and averaged online. CARVE uses a sliding 30-second window of statistics, averaged, as the input vector for both learned models.

Each of the two models predicts a single value, given this vector, which represents the expected gain or loss of a memory reallocation. These values are compared to a fixed cost. CARVE first considers the possibility of renting more memory (if possible). If the expected per-second gain from this operation exceeds the static cost of the memory, CARVE immediately initiates a reconfiguration event. If CARVE opts not to rent more memory, it instead considers the potential cost of having less memory. If this cost is less than the memory rental cost, CARVE will reconfigure the system with less memory. Although the raw system value may decrease, the overall value will increase, because the loss in value is overcome by the savings in memory cost. If CARVE determines that neither option will increase the value of the system, the configuration remains the same and CARVE continues monitoring.

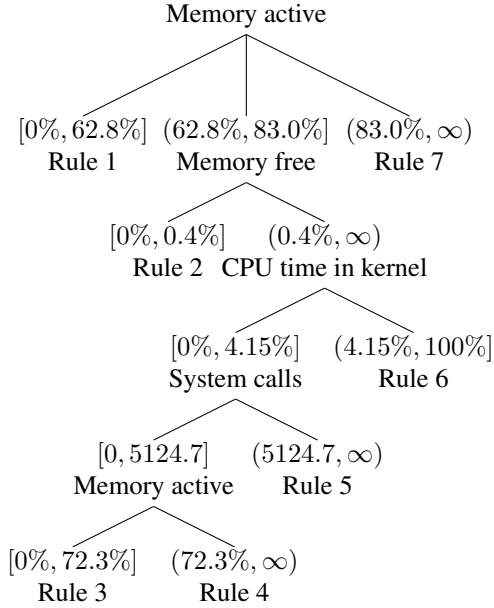
CARVE is tested with two possible costs per 512 MB block of memory: 10 and 20 units. We assume that these units are the same as in the SLA value function. The actual unit is arbitrary, so long as both are expressed in a common unit.

4. Results and Discussion

To gather a baseline for comparison of CARVE, we first run all 5 static configurations of physical memory for the database (2 GB, 2.5 GB, 3 GB, 3.5 GB, and 4GB) on the test workloads listed in Table 2. Each workload is run 15 times for each configuration, and the per-second SLA value for each run is computed. We call this the *raw SLA value* of the run. Because these static configurations represent different resource investment decisions, it is necessary to transform these into *true system values* by subtracting the per-second cost of the additional used resources. These true system values are averaged for each workload to get the overall system value for the configuration on that workload. Because no investment decisions are made online, the same data can be used for all possible resource costs by changing the cost subtracted in the calculation of the true system value.

CARVE also performs 15 runs on each workload; however, due to the cost analysis it performs, a separate set of 15 runs must be done for each cost considered. For each run, the raw SLA value is computed as before. This is

⁵The state of the web-server partition is not currently used in prediction



$$\begin{aligned}
 \text{Rule 1: SLA loss from 512 MB less} &= 0.2204 * \text{runnable processes} \\
 &- 0.0075 * \text{blocked processes} \\
 &+ 0.4771 * \% \text{ memory active} \\
 &+ 0.6544 * \% \text{ memory free} \\
 &+ 0.0489 * \text{file pages in} \\
 &- 0.0003 * \text{file pages out} \\
 &+ 0.0052 * \text{swap pages in} \\
 &- 0.0010 * \text{swap pages out} \\
 &- 0.0129 * \text{pages freed per second} \\
 &- 0.0001 * \text{system calls per second} \\
 &- 0.0113 * \text{context switches per second} \\
 &+ 0.0313 * \% \text{ CPU time in user space} \\
 &+ 7.1543 * \% \text{ CPU time in kernel space} \\
 &+ 0.8540 * \% \text{ CPU time idle} \\
 &+ 0.0395 * \% \text{ CPU time waiting for I/O} \\
 &+ 270.6244 * \text{number physical CPUs used} \\
 &- 2.3686 * \% \text{ entitled capacity used} \\
 &- 97.3591
 \end{aligned}$$

Figure 3. The tree and one regression rule learned for decreasing memory by 512 MB. Each of the rules is a simple linear function approximator that calculates a weighted sum of the numeric values of the input vector.

Random test	Phase								
	1			2			3		
	B	S	O	B	S	O	B	S	O
1	504	313	107	22	63	1089	47	669	164
2	570	531	54	163	325	545	3	58	896
3	165	224	561	160	524	447	464	674	49
4	169	862	36	916	213	23	92	361	447
5	226	142	645	2	1	934	119	728	320

Table 2. Randomly generated workloads used for testing the learner. These workloads were generated as detailed in Section 3.2.

then converted to a true SLA value; however, this conversion involves analyzing the resources actually consumed by the system during that run and subtracting the average per-second cost of the resources⁶. These 15 true SLA values are averaged to get the overall SLA value for CARVE on the workload at the given resource cost.

The results of CARVE and static configurations are shown in Figures 4 and 5, which correspond to costs of 10 and 20 respectively. Figures 6 and 7 show the amount of memory allocated by CARVE over time during each of the 5 workloads for costs of 10 and 20, respectively. Although CARVE does not always outperform all the static configurations, it is usually able to obtain comparable results. Of particular interest are the results of workload 5 with a cost of 20. We can see that in this case, CARVE significantly

outperforms all 5 static configurations.

Before the system reaches this blocked point, CARVE allocates more memory to help handle the workload. However, as the throughput drops due to the multiple EBs waiting to get access to the database, CARVE releases some memory. This is because more memory does not help to resolve this blockage (we see the same drop in throughput in all 5 static configurations). After the system resumes normal operation around 400 seconds, we can see that CARVE begins to allocate memory back to the partition, as it is needed by more clients.

Finally, to demonstrate other applications of CARVE, we simulated a system running a constant, expected workload, with a kernel memory leak imposed. This workload consisted of 500 browsing and 500 shopping users, with the system losing 56 MB of memory every 15 seconds, starting after 300 seconds into the measurement interval. This

⁶Although the resource allocation may vary greatly over the course of the run, the true SLA value is an average value over the entire run.

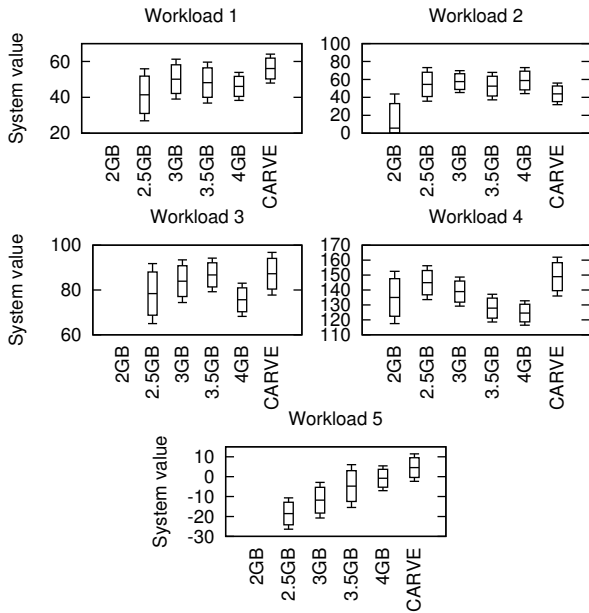


Figure 4. Results of CARVE and static configurations with a cost of 10. Boxes show the average and 95% confidence interval; the whiskers show the 99% confidence interval. Values not shown are below the range of the graph.

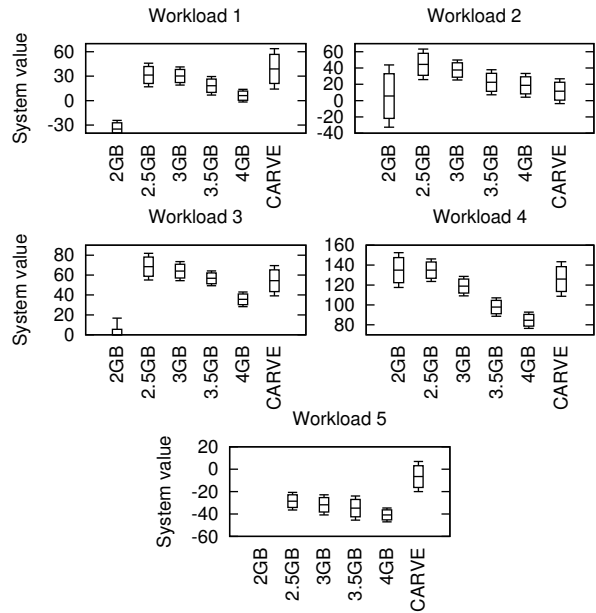


Figure 5. Results of CARVE and static configurations with a cost of 20. Boxes show the average and 95% confidence interval; the whiskers show the 99% confidence interval. Values not shown are below the range of the graph.

leak continued for 300 seconds (to a maximum of 1120 MB leaked), and then the run continued for a remaining 300 seconds. The system value over time (for a cost of 20) can be seen in Figure 8. We can see that initially, CARVE has selected a lower-memory configuration, since the additional memory does not increase overall system value. As memory becomes more scarce (starting at time 540), CARVE increases the amount of memory in use to help mitigate the memory pressure. Figure 9 shows the average memory allocation over time by CARVE for this test. In these figures, we can see that during the early stages of the run, when there is little pressure on memory, CARVE opts to remain at a low level, buying only one unit on average to have 2.5 GB of memory. As the experiment continues, we can see that there is a steady increase in memory during the 300 second leak period, with CARVE leveling off at between 3 and 3.5 GB of memory for the final 300 seconds of the run. We can see this also see this in Figure 8, where CARVE begins to degrade as the memory is leaked, but then recovers by buying more memory.

Although in this limited scenario, it would appear better to simply use the static 3.5 GB case, this is meant to illustrate an unexpected event. Because the system would likely have run for hours, if not days or weeks, before this

event, the added cost (and lower overall value) of the higher-memory static configurations would overwhelm their benefit for this unusual case. Although system values considered here are instantaneous values, the total value to the system administrator would actually be the integral of the value curve; if we extrapolate the initial values of the different configurations backwards in time, it is clear that the lower memory configurations are better. However, during this unexpected event, CARVE provides a way to recover the system gracefully, allowing for a scheduled, rather than unexpected, outage, without needing constant overprovisioning. It is also worth noting that in this case, the maximum static memory configuration (4 GB) is never a good choice, as it is always a lower system value than the 3.5 GB case. This is because the extra memory does not increase the performance of the system, while costing an extra 20 units for every second of the run.

Clearly, there is a potential benefit to be gained by using this approach. In many cases, CARVE is competitive with the optimal static configuration, showing that it is a reasonable alternative. Because the optimal configuration can vary for different workloads, CARVE allows the system to be configured for an unknown workload without the need to do prior analysis of the workload. We can see this

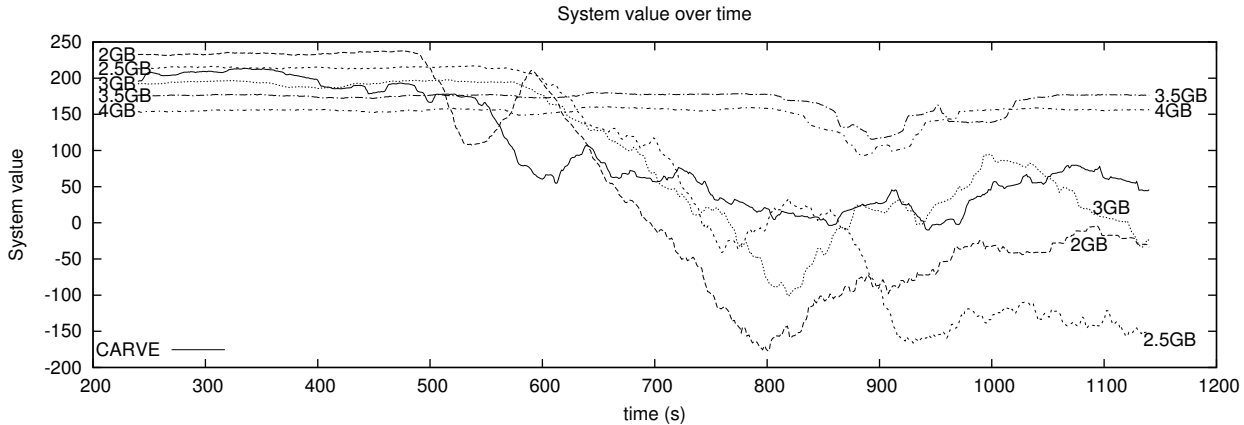


Figure 8. System value for CARVE and static configurations during a memory leak with a cost of 20. Values are calculated as the average system value used in 15 runs over a sliding 60-second window.

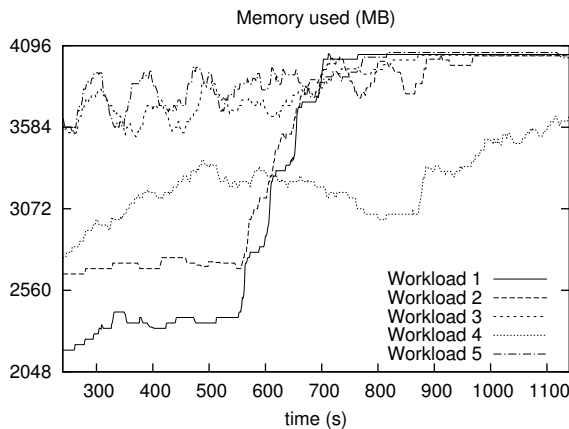


Figure 6. Amount of memory allocated to the database by CARVE over the 5 test workloads with a cost of 10. Memory amounts are calculated as the average memory used in 15 runs over a sliding 60-second window.

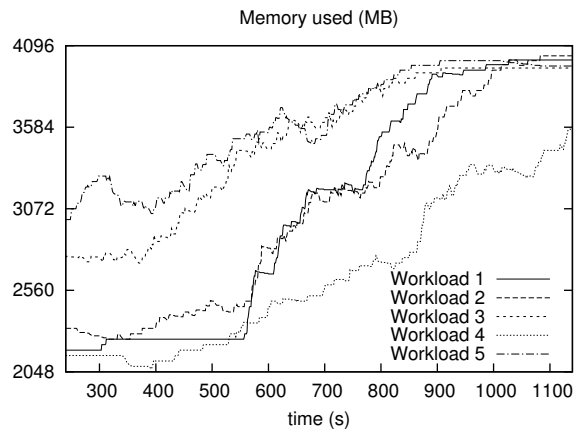


Figure 7. Amount of memory allocated to the database by CARVE over the 5 test workloads with a cost of 20. Memory amounts are calculated as the average memory used in 15 runs over a sliding 60-second window.

if we look at the average value of all configurations over all workloads, shown in Table 3. We see that, overall, CARVE outperforms all static workloads in the case where the unit cost is 10, and 4 of the 5 static workloads when the cost is 20. The one exception in this latter case is the 2.5 GB configuration; the overall better performance of that configuration is due to its substantial win on workload 2. Additionally, CARVE can help handle the situation where the system is correctly configured for an expected workload, but an unexpected memory leak affects the system.

5. Related Work

Recent work in autonomous resource provisioning has tended to fall into one of three categories. The first category

is that of maximizing utility in a cluster of homogeneous machines. In this class of work, resources are assigned at the granularity of entire machines, with no discrimination between additional processing power and additional memory.

One example of this class of work is that done by Walsh et al. [18], which explores assigning servers in compliance with a set of utility functions. The functions indicate how much utility is gained from a system as a function of demand (requests per second) and number of servers assigned; servers can also be assigned to a batch workload, whose utility is only a function of the number of servers. As the demand changes, reallocation of servers can help to maximize the overall utility of the system.

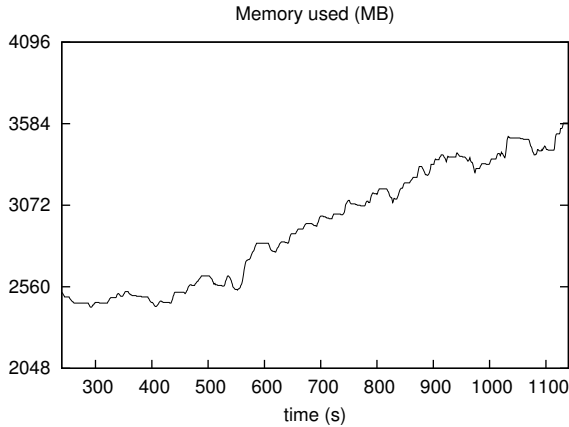


Figure 9. Amount of memory allocated to the database by CARVE during the memory leak test with a cost of 20. Memory amount is calculated as the average memory used in 15 runs over a sliding 60-second window.

Cost	Configuration	Average Value
10	2 GB	-3.0435
	2.5 GB	60.1392
	3 GB	63.7618
	3.5 GB	62.1229
	4 GB	60.8530
	CARVE	68.1557
20	2 GB	-3.0435
	2.5 GB	50.1392
	3 GB	43.7618
	3.5 GB	32.1229
	4 GB	20.8530
	CARVE	44.8645

Table 3. Average results of the different configurations over all 5 workloads at the different costs.

This work was expanded by Tesauro et al. [15] to consider multiple applications with different utility functions. In this work, a hybrid reinforcement learning is used to learn the best allocation of servers given the request rate of one server. The learned model is trained off-line with a neural network, and then continues training online to improve this initial allocation algorithm. This requires feedback from the system to indicate to the learner the value of a single transaction or group of transactions; our work uses only the aggregate value of many transaction and has no knowledge of an individual transaction.

Bennani and Menascé [2] also examine the situation of assigning servers to distinct applications. Performance models are built and used to estimate the response time of

the applications with multiple numbers of servers; the response time controls an application-specific utility function. These models and utility functions are used in a global controller that uses a beam search to find an expected optimal configuration for the current workload, which is then applied.

Chen et al. [5] implement a system to automatically add servers to the database layer of a multi-tier system using a k-nearest-neighbors approach. Although their use of system statistics is similar to this work, they consider entire machines with the goal of simply complying with an SLA, not the trade-off between additional value and resource cost.

The second category involves resource reallocation in response to power management. This work tends to involve either powering off full systems or throttling back CPU frequencies in order to move to a lower power state. One example is Kusic and Kandasamy’s work [12], which investigates balancing the cost of operating a CPU with the revenue from an SLA. The authors use queuing models and a Limited Lookahead Controller to make decisions about the operating frequency of the server’s processor. Training the queuing models involves knowledge about the system including tiers, connections, and response time at different frequencies. This work is only applicable to processors, as memory cannot be run at different frequencies.

Urgaonkar et al. [17] also use a queuing model to determine when more or less system power is needed at a given tier of a multi-tier system. They use both a predictive and reactive approach to increasing or reducing system power. In this work, the granularity of resource reallocation is at the level of entire systems, and there is no cost to bringing additional servers online, as they are assumed to be held in reserve for handling workload.

Hangs et al.[24] analyze windows of incoming workload and CPU utilization to estimate the per-transaction CPU cost of a system running a transactional workload. This data could then be used to predict the correct amount of CPU needed for a future window, although in this work, the prediction is limited to only error analysis. Additionally, this work requires knowledge of the incoming transactions, although it does not require detailed knowledge of the workload, instead inferring it from the individual transactions.

Finally, there has been some limited recent work in the space of utility computing dealing with bringing up additional servers, rather than expanding existing servers. Wang et al. [19] address the situation where applications are easily activated on a heterogeneous utility computing system, and needed capacity is addressed through additional applications. This work uses queuing models to attempt to use the minimum number of applications to maintain an expected response time. However, there is no cost associated with additional application instances, and the goal is solely to satisfy a given SLA.

6. Conclusion

As utility computing becomes more widely available and the use of virtualization solutions continues to grow, there is likely to be a growing number of businesses using these frameworks to implement their systems. Although current provisioning models tend toward system overprovisioning, this new paradigm enables administrators to provision only for an expected workload. Using the availability of on-demand resources, they can increase resource allocations if and when the additional value surpasses the resource cost. We demonstrate that an autonomous agent can make these value estimates and allocation decisions.

This paper presents CARVE, one such autonomous agent. CARVE is able to make competitive choices balancing the resource cost with the gain in value provided by this resource, when measured against an SLA which rewards the system for satisfying response time requirements and penalizes the system for violations. CARVE is also able to outperform all static configurations in some situations, and is overall generally competitive with all static configurations.

Our ongoing research involves further work to improve the accuracy of CARVE, as well as extending it to determine the best investment among multiple resources (i.e., CPU or memory). Additional directions include workload and switching cost estimation, to determine when a change should be made, as well as decisions when faced with non-static costs for resources and the ability to allocate or deallocate multiple units of resource at a time.

Acknowledgments

We would like to thank Dilma Da Silva of IBM Research for supplying the hardware used in this research. This work was supported in part by NSF awards IIS-0237699 and CNS-0615104.

References

- [1] Amazon Elastic Compute Cloud (Amazon EC2). Amazon.com, Inc., 2006. <http://aws.amazon.com/ec2/>.
- [2] M. N. Bennani and D. A. Menascé. Resource allocation for autonomic data centers using analytic performance models. In ICAC [9], pages 229–240.
- [3] A. Birman and J. J. Ritsko. Preface. *IBM Systems Journal*, 43(1):3, 2004.
- [4] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural evaluation of Java TPC-W. In *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, January 2001. Code available at <http://www.ece.wisc.edu/~pharm/tpcw.shtml>.
- [5] J. Chen, G. Soundararajan, and C. Amza. Autonomic provisioning of backend databases in dynamic content web servers. In ICAC [10], pages 231–242.
- [6] e-business on demand: the next wave of IT services. IBM Global Services, 2002. <http://www.ibm.com/services/ondemand/files/nextwave.pdf>.
- [7] D. F. Garcia and J. Garcia. TPC-W e-commerce benchmark evaluation. *Computer*, 36(2):42–48, February 2003.
- [8] IBM eServer–Capacity on Demand. International Business Machines Corporation, 2006. <http://www.ibm.com/servers/eserver/about/cod/>.
- [9] *Proceedings of the 2nd International Conference on Autonomic Computing*, Seattle, WA, June 2005.
- [10] *Proceedings of the 3rd International Conference on Autonomic Computing*, Dublin, Ireland, June 2006.
- [11] *Proceedings of the 4th International Conference on Autonomic Computing*, Jacksonville, FL, June 2007.
- [12] D. Kusic and N. Kandasamy. Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems. In ICAC [10], pages 74–83.
- [13] M. A. Rappa. The utility business model and the future of computing services. *IBM Systems Journal*, 43(1):32–41, 2003.
- [14] W. D. Smith. TPC-W: Benchmarking an ecommerce solution. Technical report, Intel Corporation, 2000. <http://www.tpc.org/tpcw/TPC-W.Wh.pdf>.
- [15] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. A hybrid reinforcement learning approach to autonomic resource allocation. In ICAC [10], pages 65–73.
- [16] Transaction Processing Performance Council. *TPC Benchmark™ W (Web Commerce) Specification*, February 2002. Version 1.8.
- [17] B. Urgaonkar, P. Shenoy, A. Chandra, and P. Goyal. Dynamic provisioning of multi-tier Internet applications. In ICAC [9], pages 217–228.
- [18] W. E. Walsh, G. Tesauro, J. O. Kephart, and R. Das. Utility functions in autonomic systems. In *Proceedings of the 1st International Conference on Autonomic Computing*, pages 70–77, New York, NY, May 2004.
- [19] X. Wang, D. Lan, G. Wang, X. Fang, M. Ye, Y. Chen, and Q. Wang. Appliance-based autonomic provisioning framework for virtualized outsourcing data center. In ICAC [11].
- [20] J. Wildstrom, P. Stone, and E. Witchel. Autonomous return on investment analysis of additional processing resources. In *2007 Workshop on Adaptive Methods in Autonomic Computing Systems*, June 2007.
- [21] J. Wildstrom, P. Stone, E. Witchel, and M. Dahlin. Adapting to workload changes through on-the-fly reconfiguration. Technical Report UT-AI-TR-06-330, The University of Texas at Austin, Department of Computer Science, AI Laboratory, 2006. <http://www.cs.utexas.edu/ftp/pub/AI-Lab/tech-reports/UT-AI-TR-06-330.pdf>.
- [22] J. Wildstrom, P. Stone, E. Witchel, and M. Dahlin. Machine learning for on-line hardware reconfiguration. In *Proceedings of the 20th International Joint Conference On Artificial Intelligence*, pages 1113–1118, Hyderabad, India, January 2007.
- [23] I. H. Witten and E. Frank. *Data Mining: Practical machine learning tools with Java implementations*. Morgan Kaufmann, San Francisco, 2000.
- [24] Q. Zhang, L. Cherkasova, and E. Smirni. A regression-based analytic model for dynamic resource provisioning of multi-tier applications. In ICAC [11].