Towards Adaptive Resource Allocation for Database Workloads

Cong Guo and Martin Karsten

David R. Cheriton School of Computer Science, University of Waterloo

{c8guo,mkarsten}@uwaterloo.ca

ABSTRACT

Modern computer systems provide hardware resources that allow database systems to execute a large number of tasks in parallel. However, no software system is perfectly scalable, and allocating more resources does not necessarily result in better performance. For commensurate resource allocation and increased efficiency, it is desirable to dynamically allocate hardware resources according to workload demands and conduct hardware consolidation. Given the complexity of database systems and their workloads, it is challenging to design such an adaptive algorithm. This paper addresses this problem using a simple feedback mechanism. The contributions of this work are twofold. First, an application-agnostic performance metric based on hardware performance counters is proposed to measure system performance online. This fine-grained metric enables agile feedback even for long running analytical workloads. Second, an allocation algorithm is presented that is designed based on fuzzy control techniques. The controller does not need a system model or prior training. Evaluation results show that a good correlation exists between the system-level metric and application-specific performance metrics. Further, a database system with our controller can achieve performance comparable to that obtained with manual tuning.

1. INTRODUCTION

Modern applications need to process large quantities of data. Database management systems (DBMSs) play an important role in these data-intensive applications, handling a large number of concurrent transactions and/or analytical queries. Modern computer systems provide hardware resources that enable database systems to execute these tasks in parallel. However, no software system is perfectly scalable, and database systems are no exception. Allocating more resources to a DBMS does not necessarily result in better performance. Beyond a certain point, the marginal performance improvement becomes very small or even negative. In a multi-tenancy scenario, a physical machine may host multiple database servers; a cloud service provider may provide data management services to multiple customers (Database-as-a-Service). It is a waste to allocate more resources if a database system has achieved optimal performance. The resource utilization of the whole system can be improved by letting the remaining resources be utilized by other database systems or other applications. Moreover, the frequency of idle CPU cores can be scaled down to reduce power consumption. Database systems and workloads are highly complex and dynamic, so it is challenging to allocate hardware resources to a DBMS to achieve optimal performance. The manual tuning process is usually time-consuming and skills-intensive for system administrators. More importantly, a static configuration hardly meets the time-varying requirements of DBMSs. This work focuses on how to allocate appropriate hardware resources to a DBMS automatically and dynamically.

We use CPU cores as a study case for adaptive resource allocation. More CPU cores allow database systems to run more threads in parallel. Database systems benefit from the consequent increase of parallelism. However, the increase of parallelism is something of a double-edged sword. Too many simultaneously executing threads may cause contention for resources like cache, memory bandwidth, and locks. For typical transactional and analytical workloads, system performance initially increases, if the number of cores increases, but starts to level off beyond a certain point.

Due to the complex nature of database systems, feedback control is used to conduct adaptive resource allocation in our approach. The control loop consists of two components: one measures the system performance online and the other one determines the number of CPU cores based on the real-time feedback and control rules. This work makes two contributions to constructing the two components of the resource management system.

First, an application-agnostic fine-grained metric named *user-level instructions per time* (UIPT) is proposed to measure workload performance online. The metric is based on hardware performance counters and provides accurate and real-time feedback to the control loop. Application-specific performance metrics like throughput and response time can be calculated or inferred indirectly from UIPT. This metric works well for both OLTP and OLAP workloads. In particular, UIPT enables performance measurement on the fly for analytical queries with long running times. UIPT is not restricted to database systems. It is an application-agnostic

metric that allows the implementation of a resource management system transparent to applications.

Second, the allocation algorithm introduces fuzzy control that incorporates both human knowledge and uncertainty. Control theory provides principles to the design of feedback loops to handle changes, uncertainties, and disturbances in database systems. Fuzzy control simplifies the controller design because no specific system model is required. The heuristics are expressed through linguistic rules. The simplicity of the algorithm allows agile responses to changes of workloads and the operating environment. Classical controllers are typically used for regulatory control, that is, to keep the performance metric at a desired reference value. The control objective for our fuzzy controller is optimization instead of regulation. A reference input is not necessary in fuzzy control, so this technique is suitable for this kind of optimization problem.

Experiments are conducted to evaluate the UIPT metric and the design of the fuzzy controller. It turns out that the metric is a good indicator of system performance for typical database workloads and some other parallel programs. The database system under control of our algorithm quickly converges to a steady state with consistent workloads and adapts to changeable workloads. The performance is close to the optimal performance with a static number of cores.

The rest of the paper is organized as follows. Section 2 describes the characteristics of database workloads and the background for the design of the controller. Section 3 introduces the UIPT metric used to measure the real-time performance of the target system. The details of the controller architecture are illustrated in Section 4. Section 5 presents the results of our evaluation experiments regarding UIPT and the controller design. An overview of related work is given in Section 6. The paper is concluded and future work is briefly discussed in Section 7.

2. PROBLEM BACKGROUND

This paper studies a situation in which a single database server runs on a multi-core machine. Available CPU cores determine how many threads run in parallel on the physical machine. For a stable OLTP workload, increasing the number of CPU cores initially increases the throughput of the system, but if too many cores are allocated the throughput levels off. Similarly, for an OLAP workload, the execution time initially decreases, but starts to increase at a certain point. This trend is verified by the experimental results with both OLTP and OLAP workloads. Figures 3 and 4 show how system performance changes with the number of cores allocated to a database system (See more details in Section 5.1). Furthermore, the workload of a database system may not be stable. Therefore, the management system should be able to identify any changes in the workload, and then change the resource allocation accordingly. Our resource management system aims to allocate a proper number of CPU cores adaptive to the demand of workloads. The design contains two components that cover the two problems respectively. The inner *fuzzy controller* looks for the optimal point for the present workload; the outer *self-adaptor* checks for workload changes and then adjusts the inner controller. The architecture of the whole controller is shown in Figure 1. This section introduces the rationale behind the control loop. A management system for other more-complex situations can be constructed based on this work.

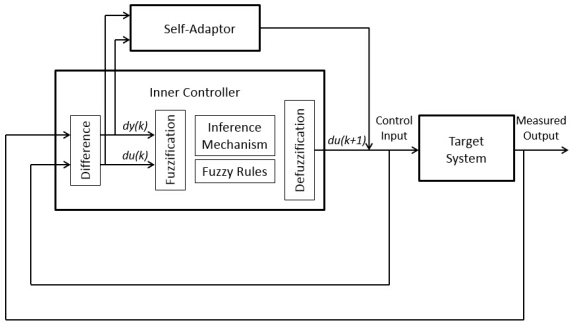


Figure 1: Architecture of the Controller

Our controller design is based on the assumption that the performance curve of database workloads is a concave curve. Take the throughput of OLTP workloads as an example. The throughput function $f(x)$ of the number of cores x can be approximated by a concave-downward curve whose derivative $f'(x)$ decreases with the increment of cores. The problem is to find the stationary point of f, which is the solution to $f'(x) = 0$, also known as the root of the derivative. As the derivative monotonically decreases, a simple linear search strategy can be used to find the stationary point. The search strategy is to increase x when $f'(x) > 0$ while decreasing x when $f'(x) < 0$. Instead of modelling the unknown $f(x)$, our approach uses $\frac{\Delta f}{\Delta x}$ to estimate whether $f'(x)$ is positive or negative. In the search, if adding cores by Δx causes Δf to become negative, the number of cores must be decreased by Δx; otherwise more cores ought to be added. The search from the opposite direction is similar. Because the number of cores is discrete, the exact stationary point may not be reached. The searching procedure will stop at the integer point closest to the stationary point. The search step (Δx) can adapt to the value of the derivative, i.e., Δx can be changed to a larger value when $|\Delta f|$ is large, because a large $|\Delta f|$ implies that the system is far from the stationary point. It should be noted, though, that the throughput curve of a complex database system is not a perfectly smooth concave curve like that of a simple quadratic function. For example, the throughput may not decline dramatically when the number of cores is larger than the stationary point, but fluctuate with a downward trend. The controller assumes that the performance curve is essentially concave, but also takes possible anomalies into consideration.

Feedback control is suitable for this kind of optimization problem. This problem could also be solved using other optimization techniques. The advantage of fuzzy control is that an accurate model of the database system is not required. Fuzzy control uses qualitative descriptions of target systems and controller actions, so it is easier to incorporate human expert knowledge on resource management in the form of fuzzy rules. Fuzzy control handles the uncertainty caused by measurement noise and system disturbances, in a natural way via fuzzy sets. Moreover, there is no specific quality of service objective in the core allocation problem. Thus, there is no reference input for the classical proportional-integral-derivative controller. In contrast, a reference input is not necessary in fuzzy control [5].

The feedback loop operates in discrete time. As shown in Figure 1, the fuzzy controller has two inputs: the change

of control input ($du(k)$) and the change of measured output ($dy(k)$) between the current and previous intervals. The output of the target system is measured in each interval, and the control input to the target system in the next interval ($du(k+1)$) is determined based on previous measurements. In our case the control input is the number of cores, and the output of the target system is the low-level system throughput UIPT. Fuzzy rules are expressed using linguistic variables in "IF-THEN" form. The qualitative rules do not contain numeric values. A fuzzification process converts numeric inputs into fuzzy sets that quantify the rules. Fuzzy sets are associated with membership functions that quantify the certainty that input values may be classified as linguistic values. The inference mechanism determines which rules are relevant and to what extent, and draws a conclusion using the rules. Finally, a defuzzification method aggregates the outcomes of all implied fuzzy rules and inversely translates the resulting fuzzy set to a single numeric value.

The inner controller is designed to find the optimal resource allocation when assuming a stable workload. The performance curve and its corresponding optimal point may shift when the workload changes. The workload changes are unpredictable in many real world applications. It is infeasible to include all environmental factors into a complicated control system. An alternative way is to restart the inner controller and find the new optimal point. Therefore, a self-adaptor is used to detect a change in the workload and restart the fuzzy controller. After the system reaches a steady state for a certain time period, the self-adaptor changes the allocation to probe whether the workload has changed. Prior knowledge about workload characteristics can be combined into this component to adjust the output or parameters of the inner controller.

3. PERFORMANCE METRIC

Real-time performance measurement is necessary for agile feedback control. Different application systems may adopt different performance metrics. For example, the transactions per second (TPS) metric is usually used to judge the performance of OLTP workloads executed by a DBMS. For OLAP workloads, the performance metric of interest is typically response time instead of throughput. However, analytical queries can have widely varying response times, from seconds to hours. Therefore, TPS cannot reflect the real-time progress of analytic workloads, and ad-hoc averages of response times do not make much sense either. Neither metric is thus suitable for online feedback measurement of OLAP workloads.

This work proposes *user-level instructions per time* (UIPT) as a real-time indicator of system performance. UIPT is calculated by dividing the number of user-level retired instructions by the measurement interval. UIPT represents low-level throughput itself, but it can also reflect the progress of workloads. Hence, UIPT can be used to indirectly estimate other performance metrics for both OLTP and OLAP workloads.

Most modern microprocessors provide hardware performance counters that can be used to monitor performance in real time. The number of retired instructions is measured during runtime. All threads of the target system are monitored, and counts are aggregated. With a highly concurrent workload, threads in a database system might wait for locks, but blocking waits typically happen in the operating system kernel. These operations do not reflect workload progress in a database system. According to profiling results in our testbed, most of the instructions retired at the kernel level belong to this category. Therefore, only user-level instructions are counted in UIPT, whereas kernel-level instructions are excluded. In UIPT, the number of user-level instructions is used to estimate the number of productive instructions. In certain (but hopefully rare) cases, if a considerable number of instructions are retired in user-level synchronization routines, thus not productive, UIPT cannot reflect the application's progress accurately.

Instructions per cycle (IPC) is a traditional metric for low-level system throughput. However, while IPC can be used to measure some form of resource utilization, it is not meaningful as a progress metric on modern multi-core platforms. CPU cycles do not represent the absolute length of time, and thus IPC does not reflect system throughput accurately. First, CPU frequencies may (independently for individual cores) change while a program is executed. Second, hardware performance counters do not necessarily count cycles when cores are idle, thus potentially overstating progress. In addition, for a multi-core machine, it is not clear whether the total or average number of cycles across cores should be used in the calculation. The Linux perf tool uses the total number of cycles to calculate IPC when it measures multiple cores. In our experiments, the correlation between IPC from the perf tool and high-level performance metrics is poor. Previous studies in the literature also show that the classical IPC metric may inaccurately reflect performance and lead to incorrect or misleading conclusions for multi-threaded programs on multi-processor systems [1]. Finally, as discussed above, not all the retired instructions should be counted when assessing application-level progress. In contrast, UIPT is calculated using wall-clock time instead of CPU cycles, and the the above issues are avoided. We conjecture that blocking time does not affect the feasibility of UIPT, but for the sake of clarity, this paper focuses on CPU-bound workloads. Experimental results in Section 5.1 show that UIPT performs better than IPC with respect to correlation with high-level performance metrics.

Our algorithm continually measures UIPT values and compares the relative changes between adjacent time intervals. This fine-grained metric allows agile and accurate reactions. In addition, UIPT is a generic metric that applies to programs other than database systems. With such a low-level metric, the controller can be implemented without modifying the database system or even used to schedule various software systems in the operating system or hypervisor. The correlation between UIPT and high-level performance metrics like throughput of transactions and execution time is studied in Section 5.1. The results show a strong correlation, thus UIPT is used to estimate high-level performance metrics.

4. CONTROLLER DESIGN

This section introduces a proof-of-concept design for the resource management system. The control system consists of the outer self-adaptor and the inner controller. When the control system initializes, the self-adaptor first detects the position of the system at the curve via a probing allocation. Next, the inner controller based on fuzzy control searches for the number of cores that is optimal for the target system. After the inner controller reaches the steady state for a

Algorithm 1 Control(pid, du)

y: measured UIPT
$prey$: previous measured UIPT
du: previous change of cores
dy: relative change of UIPT
dC: change of cores
$interval$: measurement/control interval
$period$: period that the controller keeps stable
$accum$: accumulated change of UIPT during the period
$probe$: probe flag
$decr\_threshold$: threshold deciding whether to reduce cores
$incr\_threshold$: threshold deciding whether to add cores

```
1: y ← Measure(pid, interval)
2: dy ← (y − prey)/prey
3: dC ← Fuzzy(du, dy)
4: if du = 0 then
5:     accum ← accum ∗ (1 + dy)
6:     if period is up then
7:         if accum < decr_threshold then
8:             dC ← −1
9:         else
10:            dC ← +1
11:            probe ← True
12:        end if
13:        accum ← 1
14:    end if
15: else
16:    if probe then
17:        if dy < incr_threshold then
18:            dC ← −1
19:        end if
20:    else
21:        if du < 0 AND dC = 0 then
22:            dC ← −1
23:        end if
24:    end if
25: end if
26: return dC
```

Algorithm 2 Fuzzy(du, dy)

FS: consequents of control rules
PS: certainty of consequents
gdu: gain factor for du
gdy: gain factor for dy
gdC: gain factor for dC

```
1: Fuzzification(du*gdu, dy*gdy, FS, PS)
2: dC ← AVG(FS, PS)
3: return dC*gdC
```

window of time, the self-adaptor performs another probing allocation to check whether the workload has changed. The algorithm details are illustrated in Algorithm 1 and 2 and explained in the following sections.

4.1 Inner Fuzzy Controller

The basic components of the inner fuzzy controller are shown in Figure 1. As introduced in Section 2, linguistic variables and fuzzy rules are central to fuzzy control. Three linguistic variables are used to describe two discrete inputs and one output of the inner controller. *change-in-cores* describes how the number of cores changes in the current interval (refer to $du(k)$ in Figure 1), and it takes on the two values: decrease and increase, no matter how many cores decrease or increase. *change-in-throughput*, as the name implies, is the relative change of UIPT between the current and previous intervals ($dy(k)$). This variable's value should also be "decrease" or "increase". However, in order to describe to what extent the throughput changes, this variable employs four values: decrease, decrease_large, increase, and increase_large. *next-change-in-cores* is the output to the target database system, i.e., $du(k+1)$. Like *change-in-throughput*, it also has four values. The actual change in cores may be zero as a result of multiple rules.

Table 1 shows the "IF-THEN" form fuzzy rules applied in our controller. The first four fuzzy rules are basic rules that search for the optimal point in a curve. They cover the four situations described in the search procedure in Section 2. The last four rules just enlarge the search step Δ when *change-in-throughput* is large and the system is far from the optimal point. Sometimes the increment of throughput may be very small when one additional core is allocated, but actually the system has not reached the optimal state, and adding more cores could bring obvious improvement of throughput. A large step in Rule 5 may bypass this phase. The probing allocation of the self-adaptor can also solve this problem.

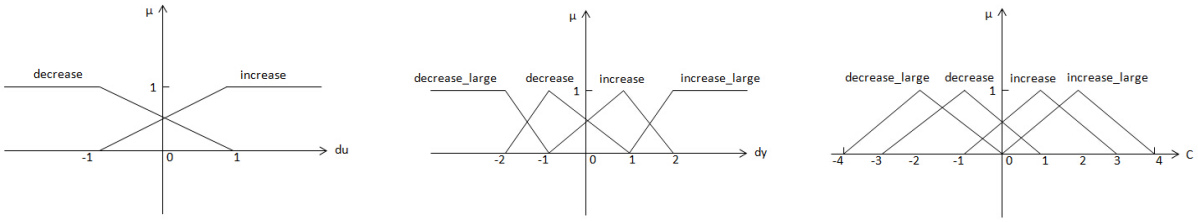
Membership functions are used to quantify all three linguistic variables. Commonly used membership functions include singleton, triangular, gaussian, sigmoidal, etc. As shown in Figure 2, triangular membership functions are used in our design. To get rid of the impact of measurement noise and system disturbances, uncertainty is introduced to describe to what extent the change of throughput is caused by the change of cores. The membership functions characterize this certainty/uncertainty. For example, in Figure 2(b), if $dy = 0.5$ then $\mu_{increase}(0.5) = 0.75$ and $\mu_{decrease}(0.5) = 0.25$, so the controller is more certain that the throughput increases. In Figure 2(a), when the actual *change-in-cores* is 0, it is considered as both "increase" and "decrease". After getting the certainty of each premise term in a rule via the membership functions, the minimum function is used to calculate the certainty of the premise of a rule, i.e., $\mu_{premise} = min(\mu_{change-in-cores}, \mu_{change-in-throughput})$.

The input values to the membership functions are not the readings from the system. All the input variables are normalized before the membership functions are applied. The measured values are multiplied by normalizing factors called gains. There are three gains in our controller, for du, dy and C, respectively. Here C is the notation for *next-change-in-cores*, i.e., $du(k+1)$ in Figure 1. The gains are denoted by g_{du}, g_{dy} and g_C, and have a significant impact on the system performance. For example, g_{dy} used in our experiments is 10, and *change-in-throughput* is one hundred percent considered as "increase_large" if the measured change is larger than 20% (see Figure 2(b)). If g_{dy} is 20, the threshold would be 10%, and the whole system will be more sensitive to *change-in-throughput*. Besides the three normalizing gains, the control interval is also an important parameter. With a shorter control interval, the system responds more quickly, but the system may become more fluctuating.

The final output of *next-change-in-cores* is the combination of the consequents of all the involved rules. The weighted average method, i.e., the AVG function in Algo-

Table 1: Fuzzy Rules

Rule	IF		THEN
	change-in-cores (du)	change-in-throughput (dy)	next-change-in-cores (dC)
1	increase	increase	increase
2	increase	decrease	decrease
3	decrease	decrease	increase
4	decrease	increase	decrease
5	increase	increase_large	increase_large
6	increase	decrease_large	decrease_large
7	decrease	decrease_large	increase_large
8	decrease	increase_large	decrease_large



(a) change-in-cores (b) change-in-throughput (c) next-change-in-cores

Figure 2: Membership Functions

rithm 2 is used for defuzzification. The equation to calculate the final output value is

$$u^* = \frac{\sum \mu_C(u) \cdot u}{\sum \mu_C(u)} \quad (1)$$

where u is the center value of each membership function in Figure 2(c) ($\pm 1, \pm 2$), and $\mu_C(u)$ is the respective certainty of that consequent. The weighted average method is only valid for symmetrical output membership functions.

A target system with this controller is stable by definition. If the workload does not change, the controller will converge at the optimal point as explained in the problem formulation. With a bounded input, i.e., a specific number of CPU cores, the output of the system is bounded. The experimental results in Section 5 also verify this point.

4.2 Self-Adaptor

In database systems, when the workload changes, the performance curve also shifts. Our design takes workload changes into consideration. Once the simple controller introduced above converges to a steady state, it cannot adapt to such changes. Our strategy is to keep the inner controller simple and stable, but using an outer self-adaptor to alter the inner controller in response to the workload changes. The outer self-adapter has two features in our design, monitoring workload changes and adjusting the parameters or output of the inner controller. In the current design, the self-adaptor just uses a probing allocation to detect whether the performance curve has shifted. The automatic adjustment of parameters such as the control interval and three gains is our future work. We currently set the parameters based on the knowledge about how UIPT changes with a fixed number of cores. It is not difficult to find a usable set of parameters, and the same parameters work well in the experiments with both TPC-H and TPC-E workloads.

The inputs to the self-adaptor include *change-in-cores* and *change-in-throughput* during the current control interval. The output is the adjusted value of *next-change-in-cores* (see Figure 1). The self-adaptor maintains status information about the inner controller and the target system. When the self-adaptor finds that the inner controller has converged for a window of time, it will start another probing allocation to check whether the workload has changed. As shown in Algorithm 1, if the accumulated change of UIPT during the window is less than a decr_threshold, the self-adaptor reduces one core. Otherwise, it tries adding one core, and if the UIPT does not increase more than an incr_threshold, this additional core will be reduced.

It is easy for the self-adaptor to find out whether the workload is declining. The UIPT decreases dramatically no matter how the number of cores is adjusted. However, when the cores allocated are not sufficient, the UIPT will not increase even though the workload increases. Hence, when the UIPT decreases dramatically, the self-adaptor reduces cores. Otherwise, it tries adding one core to the database server. If the UIPT increases, it means that the workload increases, so the inner fuzzy controller starts searching for the new optimal point. If the UIPT does not change obviously, or even decreases, the inner controller just takes back the trial allocation and returns to the original status. Moreover, there may not be an outstanding optimal point, especially taking into account the discrete characteristic of CPU cores. The database system may have similar performance with several different numbers of cores. The inner controller may converge to a sub-optimal point and allocate more cores than necessary. To improve the system efficiency, i.e., to use fewer

cores to get the same performance, the self-adaptor keeps reducing cores if the *change-in-throughput* is very small.

5. EXPERIMENTAL EVALUATION

This section evaluates UIPT as a proxy metric for application-level performance and the design of the controller using typical database workloads. We focus on the CPU-bound scenarios in the evaluation. The testbed is a multi-core server with 4 AMD $Opteron^{TM}$ 6274 processors (4x16 cores) and 64 GB memory. Experiments are run with 2 processors dedicated to the experimental system, and 2 processors used for workload generation. The operating system is Ubuntu Server 14.04 with kernel 3.2.0.

All the experiments are conducted in an exclusive environment. Linux control groups (CGroups) are used to isolate and control the hardware resources. We do not assign queries to cores explicitly in the experiments. The cpuset subsystem of CGroups specifies the cores that the database server can use. CGroups modifies only the cpu_allowed field in each thread structure, and leaves thread migration to the system scheduler. When the system scheduler schedules threads, it always chooses a CPU core that is allowed, regardless of whether the thread is created or resumed. The additional cost of this functionality is very small. Moreover, since all the data is stored in main memory and working sets of workloads considerably exceed the capacity of on-chip caches, the data locality does not have an obvious impact on the system performance in our experiments, either. To sum up, the control cost in our experiments is not significant.

The number of user-level instructions is measured via the Linux perf tool. The database server used in all the experiments is MariaDB-10.0.11[1]. Most of the system variables keep their default values except that the size of the buffer pool is set to 20 GB, and innodb_flush_log_at_trx_commit is set to 0. The buffer pool size is larger than the size of tables and indexes, and a warm-up is performed before the experiments, so that all workloads used in the evaluation are CPU-bound.

5.1 UIPT Evaluation

UIPT reflects the progress of programs and can be used to estimate performance changes of systems. This section evaluates the correlation between UIPT and application-level performance metrics with varying numbers of CPU cores. Results show that UIPT is a good proxy metric for various systems and workloads.

UIPT provides a solution for real-time feedback measurement of OLAP workloads. The correlation between execution time and UIPT is studied for analytical queries using a TPC-H-based workload [13]. The database size is 16 GB. TPC-H contains 22 queries of a business decision support system. These queries that process large volumes of data are usually time-consuming. Running all these queries serially on our testbed takes about 2498 seconds on average. We construct a synthetic workload of 20 concurrent clients. Each client submits all 22 queries to the database server serially, but in a different order. The database server employs 20 connection threads to process the queries from each client respectively. The same workload is executed repeatedly with the number of cores being increased from 2 to 32.

[1]https://mariadb.org/

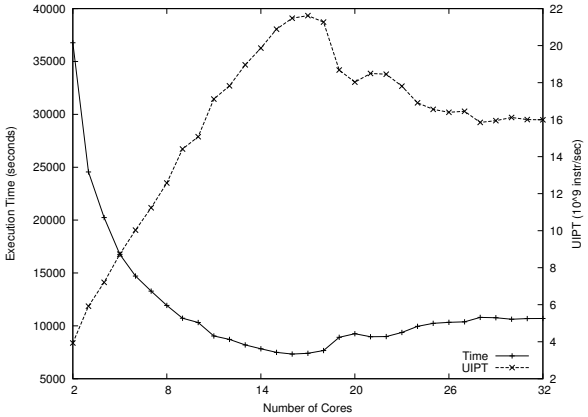


Figure 3: UIPT for TPC-H

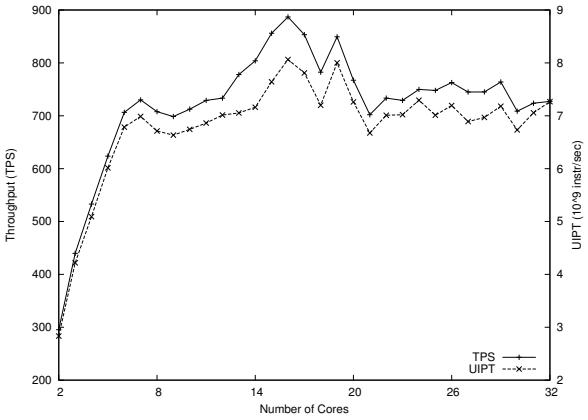


Figure 4: UIPT for TPC-E

The concurrent client sessions do not complete at the same time. Therefore, at the end of the experiments, some cores become idle while some are still busy, so the completion time of the last query does not reflect the overall time cost for the whole batch of queries. Instead, an average value is used as an estimation of the execution time in this experiment. Assume the number of cores is c. The estimation is the average of the execution time of the last c client sessions.

The Pearson product-moment correlation coefficient used in this study is $+1$ in the case of a perfect direct linear relationship, -1 in the case of a perfect inverse linear relationship. The execution time and UIPT with different numbers of cores shows a clearly inverse correlation in Figure 3. When the UIPT becomes higher, the final execution time is shorter. The correlation coefficient between time and UIPT is -0.9845 in this experiment. However, as discussed in Section 4, the relative change of UIPT compared to the last measurement is used in the controller. The correlation between time change and UIPT change with an increasing number of cores is -0.9783.

In a second experiment, the correlation between UIPT and transaction-level throughput is measured for OLTP workloads. The database and workload of TPC-E [12] is used. The total size of the database is about 8 GB. Transactions are generated using the TPC tool EGen [12]. In contrast to the standard benchmark, all types of transactions are

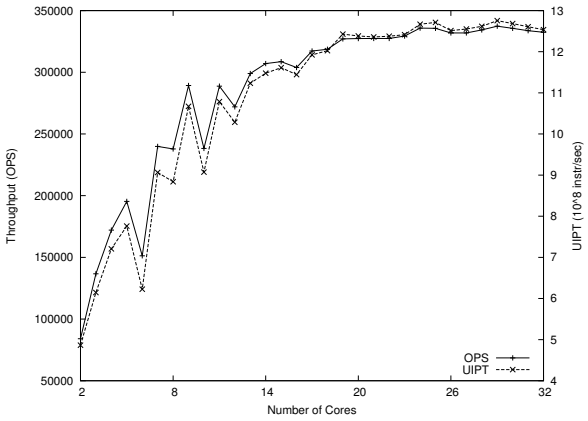


Figure 5: UIPT for Memcached

Table 2: Parsec Results

Program	Correlation between Time and UIPT	Correlation between Relative Changes
blackscholes	-0.9273	-0.9946
bodytrack	-0.9531	-0.9939
canneal	-0.9764	-0.9886
dedup	-0.9944	-0.9987
facesim	-0.9927	-0.9919
ferret	-0.8073	-0.9931
fluidanimate	-0.8994	-0.9907
freqmine	-0.8425	-0.9873
raytrace	-0.9664	-0.9979
streamcluster	-0.9604	-0.9875
swaptions	-0.9945	-0.9910
vips	-0.8635	-0.9911
x264	-0.8924	-0.9891

counted in our experiments. We use 100 user threads in this experiment. The same workload is repeated as the number of cores is varied from 2 to 32. The experiment time is 300 seconds. TPS and UIPT are measured. In the OLTP case, the variability of TPS results in repeated experiments with the same number of cores cannot be ignored. The relative change (increasing or decreasing) of TPS with different number of cores might change (see Figure 11). However, the correlation between TPS and UIPT is always close to +1 in spite of the variability the system performance. The results of one group of experiments are shown in Figure 4 as an example. It can be seen that the two metrics match very well, and the correlation is 0.9868. The correlation between the relative change of TPS and that of UIPT is 0.9878. The direct linear relationship is strong, thus the throughput of transactional workloads can be estimated using UIPT as well.

UIPT is an application-agnostic metric that can also be used for tuning of other software systems. Memcached is a distributed memory caching system for small chunks of arbitrary data. It is a key-value store whose main operations are *get* and *set*. We study the correlation between UIPT and the throughput of the memcached server (operations per second). A load generation and benchmark tool for memcached servers called `memaslap` is used in the experiment. The number of cores is varied from 2 to 32. The experiment time is 300 seconds. The ratio of get to set is 9:1, and the concurrency level is set to 128. Again UIPT shows a very strong positive correlation with the application-level throughput operations per second (OPS in Figure 5). The correlation coefficient is 0.9958. The correlation between relative changes of OPS and UIPT is 0.9527.

We expand the evaluation to various types of parallel programs. The PARSEC benchmark suite consists of 13 programs from a wide range of areas, such as computer vision, video encoding, financial analytics, animation physics and image processing [3]. In contrast to a previous study on the scalability of PARSEC benchmarks[6], our experiments use a fixed number of threads with varying numbers of cores. All programs are executed with 256 threads, except that blackscholes runs with 64 threads. The native input data set for execution on real machines is used in the experiments. The number of cores is varied from 2 to 32. The correlation results are shown in Table 2. In most cases the negative correlation between time and UIPT is very clear[2], and the correlation between relative changes of the two metrics is very obvious, too. Therefore, tuning of these programs could utilize the relative change of UIPT.

The blackscholes benchmark uses a thread barrier to make sure that worker threads do not start working until all the threads are created. The last thread enters the barrier and unblocks other threads. The number of instructions retired in this synchronization operation is proportional to the number of threads. When the number of threads is 256, the number of non-productive user-level instructions is so large that UIPT does not reflect the system performance accurately.

As a comparison, we also measure the correlation between IPC and other performance metrics for the above experiments. In TPC-H-based experiments, if the aggregated number of cycles measured using the perf tool is used for the calculation directly, IPC keeps decreasing as the number of cores increases no matter how the execution time changes. The correlation between IPC and the execution time is only 0.3301. If the average number of cycles across cores is used, the correlation is -0.8131, but still worse than UIPT. In TPC-E-based experiments, if the aggregated number of cycles is used, IPC also shows a descending trend in general, and the correlation between IPC and TPS is -0.5981 which should be positive since both of them represent throughput. If the average number of cycles is used, the correlation is 0.8336 which is much worse than UIPT. Obviously, for typical database workloads, IPC is not a very good candidate for online performance measurement.

5.2 Adaptive CPU Core Allocation

This section evaluates how the control system works with OLAP and OLTP workloads. In all the experiments, the database system with our controller performs as well as that using the best static configuration. The average number of CPU cores used is also close to the optimal value in experiments with static allocation. The simple controller does not require a complex model or manual tuning to achieve these results.

[2]If we consider the experiments with 4 to 32 cores only, for all the programs, the correlation between time and UIPT is stronger than -0.90.

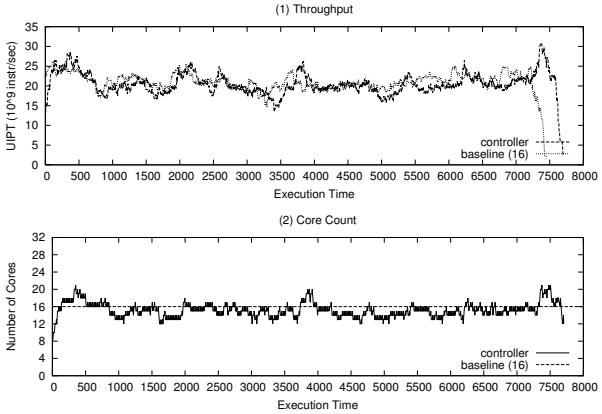


Figure 6: TPC-H: Start from 8

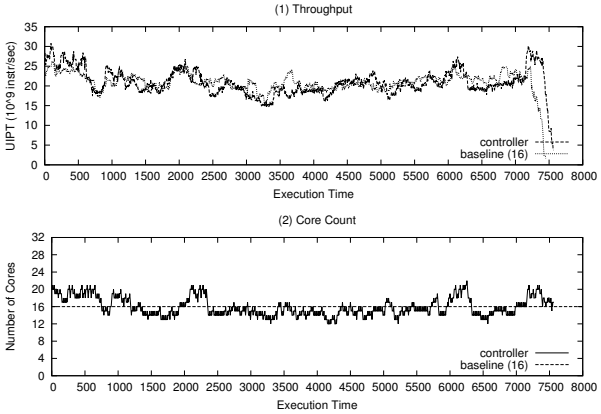


Figure 7: TPC-H: Start from 20

5.2.1 OLAP Workload

The database system and TPC-H workload used in Section 5.1 is used to evaluate how the controller works with OLAP workloads. Prior experiments with different numbers of cores show that, in the given testbed, the optimal performance of this workload is obtained with 16 cores – presumably because of NUMA effects when utilizing more than one socket. Using our controller, the performance of the database system is close to that optimal performance. Figures 6 and 7 show the execution time of each experiment and how the controller works during the execution. The control interval is 5 seconds. All threads of the database server are automatically included in the monitoring by hardware performance counters. The baseline in the two figures shows how the throughput changes when the same workload is executed with 16 cores.

Figure 6 shows an experiment with 8 CPU cores allocated to the database server at the beginning. The fuzzy controller starts the search with fewer cores than the optimal value. The execution time is 7730 seconds, only 2.7% longer than the baseline, and the average number of cores used is 15.07. In Figure 7, the number of cores allocated to the database server is 20 initially. The controller starts with more cores than the optimal value. The execution time is 7588 seconds, that is, 0.8% longer than the baseline, and the average number of cores is 15.83. In repeated experiments, the execution time with our controller is 1% to 4% longer than the average optimal time. The controller converges quickly to a steady state in all the experiments. Without any prior knowledge about the optimal number of cores, the inner controller can find the local optimal point no matter what the initial state is. It can be seen from the curves that the throughput has small fluctuations even with a fixed number of cores, but the outer self-adaptor makes the system catch the fluctuations very well.

Figure 8 shows how the controller adapts to dynamic workloads in a designed experiment. There are three phases in this experiment. The workload is changed from heavy to light and back to heavy. The heavy workload contains 20 concurrent clients, while the light workload has only 5 clients. The vertical dash lines indicate when the workload changes. Initially 8 cores are allocated to the database system. The changes of UIPT and cores are plotted in the figure. At the beginning, the number of cores increases

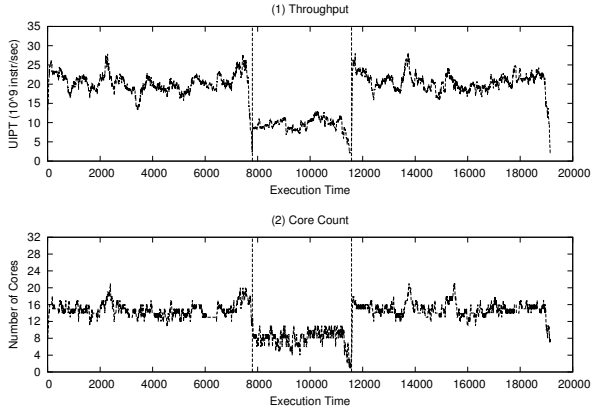


Figure 8: Changing Workloads

quickly, and so does the UIPT. The whole system reaches a steady state. When the workload switches from heavy to light, the number of cores drops and the system finds a different optimal point. When the workload changes back to the heavy one, the number of cores recovers to a higher level because more computing resources are required. The changes of UIPT and cores show that our system, particularly the self-adaptor, can identify workload changes and make adjustments accordingly. The total execution time of this experiment is only 3% more than the sum of optimal execution times of three workloads with best static configurations.

5.2.2 OLTP Workload

The TPC-E workload and configuration used in Section 5.1 is used in the evaluation of the controller for transactional workloads. As in the OLAP experiments, two groups of experiments starting with 8 and 20 cores are conducted respectively. The execution time is 300 seconds in all the experiments. The control interval is still 5 seconds. The measured UIPT values and the responses of the control system in two experiments are shown in Figure 9 and Figure 10. The baseline in the two figures shows the throughput of the same workload executed with 16 cores. The throughput of the baseline is 873 TPS. The throughput of the experiment in Figure 9 is 863 TPS, and the average number of cores used

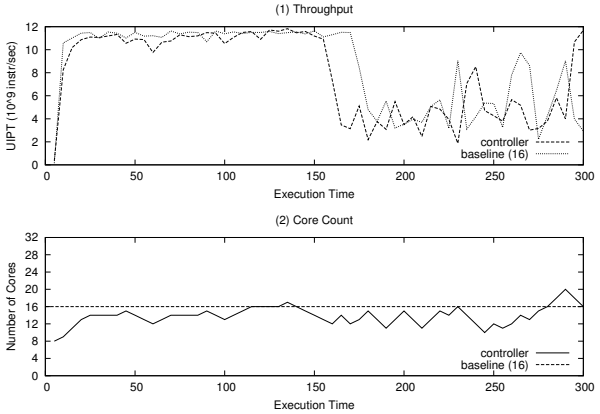


Figure 9: TPC-E: Start from 8

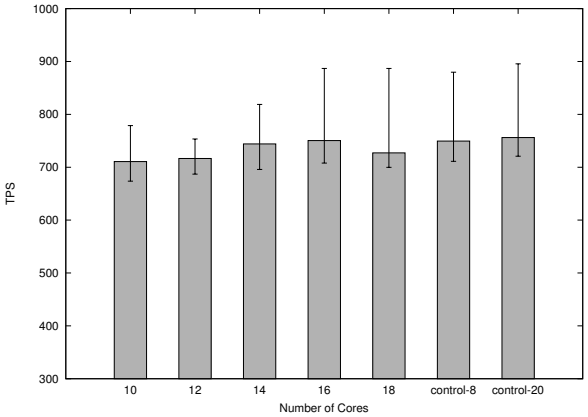


Figure 11: TPC-E Results Comparison

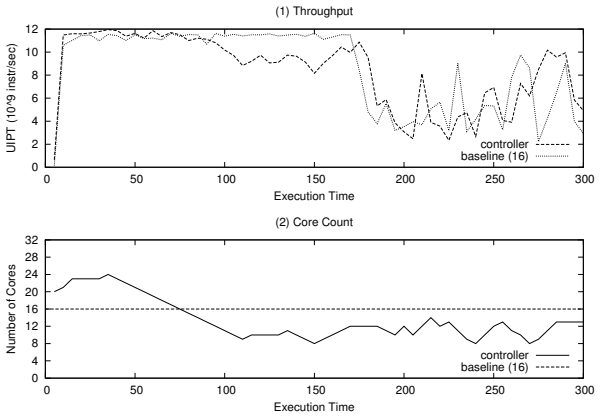


Figure 10: TPC-E: Start from 20

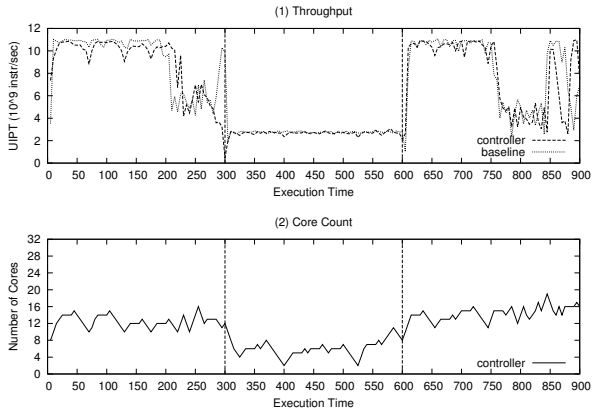


Figure 12: Changing Workloads

is 13.85. The throughput of the experiment in Figure 10 is 886 TPS, and the average number of cores used is 13.49. The UIPT measurement shows that at the beginning of the execution the throughput is high and stable. Our controller quickly converges to the optimal point no matter what the initial allocation is. However, even when the configuration of the workload does not change, large throughput fluctuations occur at the late stage. The baseline experiment shows that the fluctuation is not caused by the controller. The results confirm that the fluctuation does not prevent the controller from achieving good performance.

We do not directly compare the average results of dynamic and static allocation since the variability of TPS in repeated experiments is large. Instead, the minimum, median and maximum TPS values in 20 repeated experiments with different configurations are presented in Figure 11. Without too much prior knowledge, the controller can achieve performance comparable to that obtained with best static configurations.

A three-phase experiment is used to test how the controller works with changes of transactional workloads. The workload is changed from heavy to light and back to heavy, and each phase is 300 seconds. The heavy work load has 100 concurrent user threads, while the light workload has only 4 users. Initially 8 cores are allocated to the database system. The changes of UIPT and cores in Figure 12 show how the controller performs. The vertical dash lines indicate when the workload changes. The throughput fluctuation still takes place for the heavy workload. As a comparison, we plot a baseline result with fixed numbers of cores. The heavy workload is executed with 16 cores at the first and last phase, while 7 cores are allocated to the light workload. The inner controller converges within each phase while the self-adaptor keeps watching and enables the controller to react quickly to workload changes. The number of transactions completed at each phase reaches 99% of the corresponding baseline result. The TPS of the second heavy workload is not as good as that of the first heavy workload even though they have the same configuration. The baseline in Figure 12 confirms that the degradation is not caused by the controller.

6. RELATED WORK

Hardware performance counters provide valuable information about system status. Previous work has shown how hardware performance monitoring can be used in a fine-grained feedback loop for dynamic optimizations. Hardware performance events are used to determine cache partitioning and detect cache pollution and sharing patterns among threads in a multicore-aware operating system [2]. A low-level throughput metric similar to UIPT, named micro-operations/microsecond, is proposed in [9]. Automatic CPU

frequency scaling for Message Passing Interface (MPI) programs is done based on this indicator of CPU load. Whether this metric can be used in other scenarios is not studied in that paper. CPU utilization is used as the feedback metric in adaptive resource provisioning for virtual machines [7]. However, in the scenarios studied in this paper, the CPU utilization is high most of the time, and the variation is not large enough for a feedback controller to detect performance changes.

The classical multi-programming level (MPL) problem of database systems [11] is different from the hardware resource allocation problem studied in this paper. The basic objective of the MPL problem is to determine the proper number of transactions allowed in the database given a fixed set of resources, whereas this work allocates resources dynamically to match the demands of database workloads.

Predictive modelling is used for automatic resource management for database systems in previous work [15]. However, a good prediction model requires a well-designed learning algorithm and sufficient empirical data. Moreover, a prediction model for a system with volatile workloads, e.g., a cloud database, might be too complex for humans to understand. It is difficult to maintain and improve such a system.

Classical control theory has been applied to database self-management problems [8]. Multi-layer controllers are used for adaptive resource management for virtual machines and web servers [7, 10]. Classical control theory is used for performance regulation in these works. In some scenarios, an optimization problem is transformed into a regulation problem, so that the classical feedback control can be applied [8]. However, many database problems focus on optimization rather than regulation, and fuzzy control is a better choice.

Fuzzy control is used for the automatic tuning of the Apache web server in [4]. This work considers only a single performance curve, whereas we introduce a self-adaptor to deal with the workload changes and consequent shift of performance curves. Adaptive fuzzy modelling can be combined with prediction techniques for automatic resource management [14], whereas our simple feedback controller does not need the workload-to-resource model based on machine learning.

7. CONCLUSION

This paper proposes a novel application-agnostic performance metric for online performance measurement and an allocation algorithm based on fuzzy control that incorporates human knowledge and uncertainty. Experimental results show that UIPT is a generic metric that works well in various scenarios, and fuzzy control is a promising technique for managing hardware resources for complex computer systems. The algorithm based on fuzzy control is simple and reflects human intuition directly. The performance of the inner controller can be improved by tuning parameters such as the measurement interval and normalizing gains. Combined with other techniques, the self-adaptor can do the tuning automatically according to workload changes. A management system that allocates and configures hardware resources between database workloads with different priorities can be constructed based on this work.

Techniques in this paper can also apply to the balance between power efficiency and application performance. Modern hardware offers fine-grained and dynamic power management like Dynamic Voltage and Frequency Scaling (DVFS). Power consumption can be measured and controlled online as well. Software management problems, e.g. the MPL problem may also benefit from the search based on feedback control. Study of these relevant problems is also part of our future work.

8. ACKNOWLEDGEMENTS

This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

9. REFERENCES

[1] A. R. Alameldeen and D. A. Wood. IPC Considered Harmful for Multiprocessor Workloads. *IEEE Micro*, 26(4):8–17, 2006.

[2] R. Azimi, D. K. Tam, L. Soares, and M. Stumm. Enhancing Operating System Support for Multicore Processors by Using Hardware Performance Monitoring. *ACM SIGOPS Operating Systems Review*, 43(2):56–65, 2009.

[3] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[4] Y. Diao, J. L. Hellerstein, and S. Parekh. Optimizing Quality of Service Using Fuzzy Control. In *Proceedings of the 13th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management: Management Technologies for E-Commerce and E-Business Applications*, pages 42–53. Springer-Verlag, 2002.

[5] J. L. Hellerstein, Y. Diao, S. Parekh, and D. M. Tilbury. *Feedback Control of Computing Systems*. John Wiley & Sons, 2004.

[6] O. Itzhak, I. Keidar, A. Kolodny, and U. C. Weiser. Performance Scalability and Dynamic Behavior of Parsec Benchmarks on Many-Core Processors. The 4th Workshop on Systems for Future Multicore Architectures, http://sfma14.cs.washington.edu/ (accessed June 15, 2015), 2014.

[7] E. Kalyvianaki, T. Charalambous, and S. Hand. Adaptive Resource Provisioning for Virtualized Servers Using Kalman Filters. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 9(2):10:1–10:35, 2014.

[8] S. S. Lightstone, M. Surendra, Y. Diao, S. Parekh, J. L. Hellerstein, K. Rose, A. J. Storm, and C. Garcia-Arellano. Control Theory: a Foundational Technique for Self Managing Databases. In *Data Engineering Workshop, IEEE 23rd International Conference on*, pages 395–403. IEEE, 2007.

[9] M. Y. Lim, V. W. Freeh, and D. K. Lowenthal. Adaptive, Transparent CPU Scaling Algorithms Leveraging Inter-node MPI Communication Regions. *Parallel Computing*, 37(10-11):667–683, 2011.

[10] P. Padala, K.-Y. Hou, K. G. Shin, X. Zhu, M. Uysal, Z. Wang, S. Singhal, and A. Merchant. Automated Control of Multiple Virtualized Resources. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*, pages 13–26. ACM, 2009.

[11] B. Schroeder, M. Harchol-Balter, A. Iyengar, E. Nahum, and A. Wierman. How to Determine a

Good Multi-programming Level for External Scheduling. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, pages 60–71. IEEE, 2006.

[12] TPC-E Benchmark. Transaction Processing Performance Council. http://www.tpc.org/tpce/ (accessed June 15, 2015).

[13] TPC-H Benchmark. Transaction Processing Performance Council. http://www.tpc.org/tpch/ (accessed June 15, 2015).

[14] L. Wang, J. Xu, M. Zhao, Y. Tu, and J. A. Fortes. Fuzzy Modeling Based Resource Management for Virtualized Database Systems. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), IEEE 19th International Symposium on*, pages 32–42. IEEE, 2011.

[15] P. Xiong, Y. Chi, S. Zhu, H. J. Moon, C. Pu, and H. Hacigumus. Intelligent Management of Virtualized Resources for Database Systems in Cloud Environment. In *Proceedings of the 27nd International Conference on Data Engineering (ICDE)*, pages 87–98. IEEE, 2011.