Memory-Aware Sizing for In-Memory Databases

Karsten Molka*†, Giuliano Casale*, Thomas Molka†, Laura Moore†
*Department of Computing, Imperial College London, United Kingdom
{k.molka13, g.casale}@imperial.ac.uk
†SAP HANA Cloud Computing, Systems Engineering, Belfast, United Kingdom
{karsten.molka, thomas.molka, laura.moore}@sap.com

***Abstract*—In-memory database systems are among the technological drivers of big data processing. In this paper we apply analytical modeling to enable efficient sizing of in-memory databases. We present novel response time approximations under online analytical processing workloads to model thread-level fork-join and per-class memory occupation. We combine these approximations with a non-linear optimization program to minimize memory swapping in in-memory database clusters. We compare our approach with state-of-the-art response time approximations and trace-driven simulation using real data from an SAP HANA in-memory system and show that our optimization model is significantly more accurate than existing approaches at similar computational costs.**

***Index Terms*—Optimization; In-memory Databases; Performance; Closed Queueing Networks; Approximation, SAP HANA.**

I. Introduction

In-memory database systems leverage new technologies, such as SDRAM, flash storage, FPGAs and GPUs, to sharply optimize database throughputs and latencies. Case studies show that in-memory databases can achieve tremendous speedups, outperforming traditional disk-based database systems by several orders of magnitude [1]. As a result, in-memory systems are in high commercial demand, in particular as part of cloud sofware-as-a-service offerings [2]. This poses new challenges regarding the management of these applications in cloud infrastructures, since there is virtually no architectural design, sizing and pricing methodology focused explicitly on in-memory technologies.

This paper tackles this problem by introducing a novel provisioning framework specifically tailored to in-memory databases. We propose a novel optimization-based methodology to provision these systems at a reference timescale, minimizing costs. Our methodology can be applied to in-memory database clusters that are continuously monitored and feed performance measurements into our framework. Our framework enables what-if analyses for various in-memory database configurations regarding performance and cost without the need to setup experiments physically. In particular, we seek for load-dispatching routing probabilities that can load balance in-memory instances for a set of clients respecting the service level agreement (SLA) in place with the customer.

We use a queueing modeling approach to describe the levels of contention at resources, in order to establish the likelihood that a sizing configuration will comply to SLAs. In particular, since in-memory systems are memory-bound applications, it is crucial that their sizing models can capture memory constraints, as memory exhaustion and swapping are more likely to happen in this class of applications. Conversely, existing sizing methods for enterprise applications have primarily focused on modeling mean CPU demand and request response times. Memory occupation is difficult to model as it requires the ability to predict the probability of a certain mix of queries being active at a given time. However, probabilistic models tend to be expensive to solve, leading to slow iteration speed when used in combination with numerical optimization. To cope with this issue, we introduce a framework based on approximate mean-value analysis (AMVA), a classic methodology to obtain performance estimates in queueing network models [3]. We observe in particular that current AMVA methods are unable to correctly capture the effects of variable threading levels in in-memory database systems and propose a correction that markedly improves accuracy. Our approach retains the same computational properties of AMVA and it is simple and inexpensive to integrate in optimization programs.

We also demonstrate that multi-start interior point methods and evolution strategies can be effectively used to solve the resulting optimization programs, offering different tradeoffs between accuracy and scalability. In particular, we propose a simple yet fast mutation function for our evolution strategy that turns out to be competitive with interior point methods. Finally, we validate our approach using real traces from a commercial in-memory database appliance, SAP HANA [4].

The remainder of this paper is organized as follows. Section II motivates our research objective and gives the problem statement. Section III introduces the system characteristics of our in-memory database. A novel response time approximation is developed in Section IV, combined with a non-linear optimization program in Section V and evaluated in Section VI by numerical tests. Finally, Section VII outlines related work, while Section VIII concludes this paper and gives future work.

II. Motivation and Problem Statement

In-memory databases are a completely new type of big data analysis systems capable of processing heavily memory intensive workloads in a parallel fashion. Their resource management is a complex and difficult task that includes memory-aware sizing of these systems across heterogeneous clusters. Analytical performance models of in-memory databases can support these sizing decisions by enabling what-if analyses

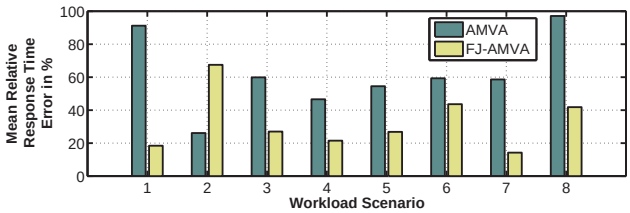


Fig. 1. Relative Response Time Error compared with Simulation

for various hardware configurations. It is therefore essential to develop models that are able to capture the behavior of in-memory databases across several dimensions. In particular, performance measures such as response times and throughputs are key metrics of in-memory systems that need to be modeled accurately. However, the extensive and variable threading-level we are faced with cannot be correctly captured by existing analytical approaches, such as AMVA [3], widely used to model the performance of multi-tier applications [5], and state-of-the-art techniques, i.e. fork-join AMVA (FJ-AMVA) [6]. To demonstrate this, we parameterized these two methods from real traces of our in-memory database SAP HANA and compared their response time predictions with a validated in-memory database simulator [7]. We give an excerpt of our results in Figure 1, which depicts the relative response time error of AMVA and FJ-AMVA compared with our simulator under different workload scenarios. We observe that using both AMVA and FJ-AMVA can result in prediction errors of more than 50%. We will therefore develop a new performance model that captures the characteristics of in-memory databases more accurately.

Our second challenge is coined by a capacity planning problem, assigning resources to in-memory databases subject to memory and utilization constraints. Optimizing memory occupation for such systems can be computationally expensive and can introduce local optima due to non-convexity. Hence we address this by proposing intelligent optimization strategies and combine these with our new performance model.

Summarizing, our main contributions are:

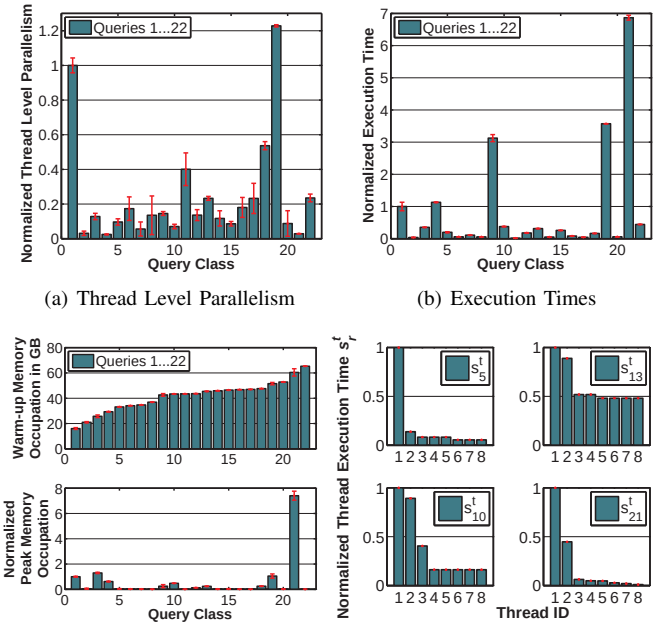
- A novel analytic reponse time approximation for in-memory databases that considers thread-level fork join
- An optimization-based formulation for seeking load-dispatching routing probabilities to minimize memory swapping for such systems subject to resource constraints
- An experimental validation that reveals the applicability of local and global search strategies
- Parameterization and evaluation of our models with real traces of an in-memory database system

To the best of our knowledge there are no methods that combine thread-level fork join models with non-linear optimization of in-memory databases, and thus represents a novelty in this research area.

III. In-memory Database Characteristics

A. OLAP Workload Characteristics

In-memory databases are optimized to execute analytical business transactions, i.e. OLAP. These types of transactions



(a) Thread Level Parallelism (b) Execution Times

(c) Memory Occupation (d) Thread Times for $t \leq 8$

Fig. 2. Workload Characteristics normalized by Query Class 1

represent read-only workloads and can thus be entirely processed in main memory. Due to their analytical nature OLAP workloads are not only computationally intensive but also show high variability in their threading levels. To emphasize these diverse characteristics, we analyzed trace logs obtained from benchmarking experiments running SAP HANA on an IBM X5 4-socket database server configured with 1TB main memory [8]. The benchmark was run with a scale factor of 100x and comprised a set of 22 OLAP queries introduced by SAP-H, an extension to the TPC-H benchmark with emphasis on analytical processing. We provide the results of our trace log analysis for all 22 query classes in Figure 2(a)-(c). All values are obtained from isolated query runs, normalized by class 1 for confidentiality and shown with their respective standard deviations. In Figure 2(a) we present the average number of CPU cores used by each query class and denote this with thread level parallelism. As expected, we see a strong variability of the parallelism across all query classes, which can increase contention for resources under OLAP workload mixes. This attains further distinction due to the varying computational expense of OLAP queries, depicted in Figure 2(b). In addition, we reveal the memory intensive character of OLAP workloads in Figure 2(c) by showing the physical memory temporarily occupied during the processing of queries, which varies on gigabyte scale. To emphasize the importance of compression during the execution of OLAP workloads, we demonstrate in Figure 2(c) that our corresponding benchmark dataset with a size of 1.3 TB was reduced to approximately 65GB after a warm-up run for each query class.

B. Request Handling and Demand Characterization

Query planning and execution are important stages during the processing of OLAP workloads. The first stage involves a query planner analyzing the query structure and creating an

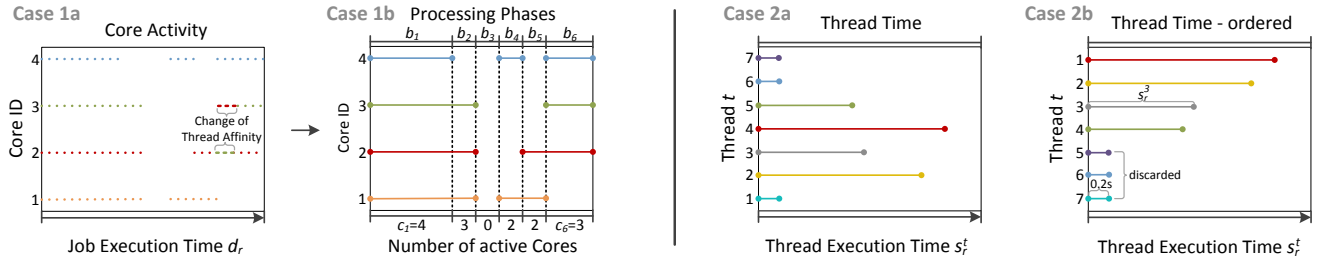


Fig. 3. Service Demand Estimation for an OLAP Query

appropriate execution plan. In the second stage, queries are executed depending on their assigned execution plan, which defines the number of threads to be requested from an internal thread pool in order to service a query. All threads pertaining to a query process are then assigned to processing cores for execution and are synchronized before a query can leave the system. Kraft et al. [7] captured this behavior from our traces to parameterize their in-memory database simulator. We therefore review this process briefly and subsequently extend it for use with our analytical model. Since our traces contain information from isolated runs for all 22 available query classes, also denoted as job classes, we can estimate two important model parameters on a per-class basis. In particular, we are considering the service demand d_r and the parallelism l_r. While d_r accounts for the average time required by our in-memory system to service one job of class r, l_r describes the number of CPU cores used on average by query class r.

In Section IV we want to compare our performance model with FJ-AMVA and the simulator developed in [7]. However, all three approaches require a different representation of d_r. Hence, we will show in the following how to extract d_r appropriately. To better illustrate this process, we represent our traces in Figure 3 by an exemplary job that consists of 7 threads and is executed on a 4-core system. Figure 3 Case 1a shows the core activity, which was sampled during the execution of our job. We see that over time, all 4 cores were differently utilized, i.e. attributable to stalling threads or changes in thread affinity. Based on the sampled core activity, we divide the execution process of a query into P processing phases, as illustrated in Case 1b. Each processing phase is defined by its duration b_p and its number of active processing cores c_p, i.e. 4 active cores in processing phase 1 and no active cores in processing phase 3. As mentioned above, the extraction of processing phases and active cores was done in [7], since their simulator requires the parameters b_p and c_p. However, we extend this process for use with our analytical model and determine d_r and l_r as aggregates of these measurements, $d_r = \sum_{p=1}^{P} b_{pr}$ and $l_r = {d_r}^{-1} \sum_p b_{pr} c_{pr}$.

In addition to the core activity, our traces record the number of threads J_r pertaining to a class r job execution process as well as the execution times of each individual thread, excluding the duration in which a thread was not active. This information was not considered by [7], and thus prompted us to extract it from the raw traces. We illustrate this in Figure 3 Case 2a, which lists all 7 threads that belong to our exemplary job. We denote the execution time of each thread t pertaining to a job of class r with s_r^t and since FJ-AMVA specifically requires this representation, we will use s_r^t for its parameterization in our experiments. Additionally, FJ-AMVA assumes that $J_r \leq I$. However, for some classes and also for our example, with $J = 7$ and $I = 4$, this is not the case. Hence, we sort s_r^t and use only the first $t \leq I$ longest running threads, shown in Case 2b. We justify this, as for the majority of classes in our traces, where $J_r > I$, $s_r^t \leq 0.2s$ for $t > I$. This means that these threads were not sampled accurately, since [8] used a sampling interval of 0.2 seconds.

IV. Fork-Join Model

In this section, we study queueing performance models for in-memory databases based on multi-core processors. In addition, we propose an efficient analytical approximation to these and present all relevant notation in Table I

A. Modeling an in-memory Database

Performance models for in-memory databases need to be aware of the complexity introduced by OLAP workloads and require a contention model that accurately captures hardware and application characteristics. Emphasized by the high level of query parallelism shown in Figure 2(a), fork-join queues prove to be an appropriate choice for modeling an in-memory database. We therefore apply fork-join queues to model the processing cores of such a system. In particular, we consider processor sharing (PS) queues in the sense of Baskett et al. [9], i.e., where service times are i.i.d. generally distributed. We employ multiclass closed queueing networks (QN) with a think time model that represents an abstraction of client think time and inter-activation times of worker threads in the database, which are dependent on the admission buffer and thread pool size. In Figure 4 we present our queueing model. It captures the behavior of jobs split into several tasks on arrival at the system, which are then assigned to processing cores in a probabilistic manner. This includes the synchronization aspect of parallel siblings at the join point and the return to the think time buffer once a job is completed.

Approaches to solve this type of QNs via simulation, e.g. in [7], emphasize the difficulty in finding analytical solutions. We will therefore discuss available approximations to QNs, before we introduce our novel analytical response time correction to fork-join queues.

B. Approximations to Fork-Join Queues

The widely used exact analytical solution for closed QNs, known as mean-value analysis (MVA), determines the re-

TABLE I
MAIN NOTATION

Symbol	Description
Workload Parameters	
R	Number of query classes
b_p, c_p	Length of processing phase p and number of active cores during p
d_{ir}, s_{ir}	Service demand and service time of class r at queue i
l_r	Number of cores used on average by class r (average degree of parallelism)
s_r^t	Service time of thread t of class r
J_r	Number of threads per class r
$\vec{N}$	Population vector with number of jobs per customer class: $N_1, \ldots, N_R$
$\vec{Z}$	Vector of per-class think times $Z_1, \ldots, Z_R$
Additional Parameters	
I_i	Number of available processing cores at server i
p_{ir}	Probability of class r jobs being routed to queue i
Performance Measures	
X_{ir}	Per-class throughput at queue i
W_{ir}	Per-class residence time at queue i
A_{ir}	Queue length at arrival instant of class r at queue i
Q_{ir}	Per-class queue length at queue i
U_{ir}	Per-class utilization of queue i
M_{ir}	Per-class memory utilization at server i

sponse time W_{ir} for a job of class r at queueing center i depending on the total number of per-class jobs $\vec{N}$ in a system as follows [10]:

$$W_{ir} = d_{ir}\left(1 + A_{ir}(\vec{N})\right). \tag{1}$$

Here, the definition of W_{ir} includes the queueing time and the service demand $d_{ir} = v_{ir}s_{ir}$, the product of per-class service time s_{ir} and visits v_{ir}. The arrival instant queue $A_{ir}(\vec{N})$ counts for the total number of jobs queuing or being serviced at i at the arrival instant of a job of class r. Based on the arrival theorem for closed QNs, $A_{ir}(\vec{N})$ can be expressed as $Q_{ir}(\vec{N}-1_r)$, which designates the queue length with one class r job less. MVA is applied in recursive fashion, but despite being analytical it gets intractable for problems with more than a few customer classes. This is addressed by Bard-Schweitzer [3], proposing an approximate MVA (AMVA) that employs a fixed-point iteration and estimates A_{ir} via linear interpolation:

$$A_{ir}(\vec{N}) \approx \frac{(N_r - 1)}{N_r} Q_{ir}(\vec{N}) + \sum_{s=1, s\neq r}^{R} Q_{is}(\vec{N}). \tag{2}$$

Synchronization in fork-join queues introduces temporal delays that cannot be described with the above product-form models. As MVA and AMVA are not applicable in that case, more recent approaches tried to address this aspect [6], [11]. Alomari et al. [6] propose a response time approximation called FJ-AMVA that sorts per-class residence times in descending order and scales them by an appropriate coefficient for better estimation of the synchronization overhead. This approach assumes s_{ir} to be the mean of the exponentially distributed service times $\tilde{S_{ir}}$. It can be shown that if s_{ir} are the same at every queue for a particular class r, $\max_i(s_{ir}) \times H_{J_r}$ equals $s^* = E[\max_i(\tilde{S_{ir}})]$, where s^* becomes the maximum service time of a job and $H_{J_r} = \sum_{j=1}^{J_r} j^{-1}$ denotes the jth harmonic number for job class r. In the heterogeneous case, s_{ir} can vary across the queues for each job class, which results in less synchronization time. FJ-AMVA approximates this by multiplying W_{ir} with $1/i$ instead of H_{J_r}.

However, the fork-join approximations in [6], [11] are less suitable for our model, as both assume exponential distributions of s_{ir}. By contrast, our service times s_{ir} and s_r^t show a generally low variability. We point this out in Figure 2(b) and 2(d), by listing the per-class execution times and their standard deviations as well as the first 8 longest running threads for a subset of our query classes. This justifies the need for a response time correction that does not rely on exponential service times.

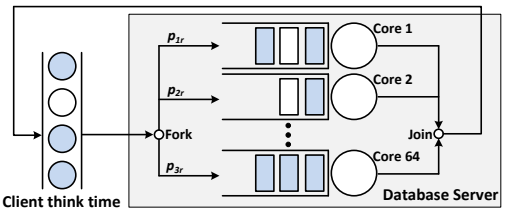


Fig. 4. Multiclass Fork-Join Queueing Model of a Database Server

C. Response Time Correction

Thread-level fork-join cannot be expressed with (1). We therefore propose an analytical response time correction called TP-AMVA, which considers the placement of tasks in fork-join queues. We assume equal probabilities of jobs being routed to a particular queue and consider jobs to not cycle within the fork-join construct. Hence, $d_r = v_r s_r = s_r$. In addition, we approximate the fork-join construct with only one single queue, which decreases processing time and simplifies its integration into our optimization program. Our correction has the following form:

$$W_r = d_r\left(1 + \sum_{s=1}^{R} Q_s\, \delta_{rs}\, \frac{l_s}{I}\right), \tag{3}$$

where the response time W_r is calculated as the service demand d_r inflated by a factor that represents the service rate degradation under processor sharing due to jobs, which already compete for resources at the same queue. The arrival queue A_s is estimated by employing Bard-Schweitzer:

$$\delta_{rs} = \begin{cases} \dfrac{N_r - 1}{N_r}, & s = r \\ 1, & s \neq r, \quad \forall s, r. \end{cases} \tag{4}$$

Since we record thread-level information for each query class, we are able to better approximate the fork-join feature. For this we correct A_s by the factor l_s/I to estimate the per-core queue length in a system with I cores. The performance measures W_r, throughput X_r and Q_r can then be resolved by employing the AMVA fixed-point iteration. In addition we approximate the utilization in a fork-join system with:

$$U = \sum_{r}^{R} U_r \frac{l_r}{I}. \tag{5}$$

Before we evaluate this model, we present an alternative approximation to (3), which is an empirical calibration. It follows the idea that an arriving class r job affects W_r depending on its probability p_r being routed to a particular queue in the fork-join construct. Hence, we correct the class r queue length Q_r by multiplying with p_r:

$$W_r = d_r\left(1 + \sum_{s=1}^{R} Q_s\, \delta_{rs}\, \frac{l_s}{I}\, p_{rs}\right), \tag{6}$$

TABLE II
RELATIVE ERROR (%) COMPARED WITH SIMULATION FOR SCENARIO S_i

Method	S_1	S_2	S_3	S_4	S_5	S_6	S_7	S_8
AMVA with $(l_r/I) \times s_r$	91.2	26.1	59.9	46.6	54.5	59.3	58.6	97.1
AMVA with s_r	3166.6	471.9	127.3	97.5	367.3	554.3	277.4	649.2
FJ-AMVA	18.4	67.5	27.0	21.5	26.8	43.6	14.2	41.8
probabilistic TP-AMVA	1.8	5.9	11.0	7.9	18.1	3.9	10.6	5.1
static TP-AMVA	4.3	18.2	8.2	11.1	20.4	11.2	12.4	3.9

where p_{rs} is defined as:

$$p_{rs} = \begin{cases} \frac{l_r}{I}, & s = r \\ 1, & s \neq r, \quad \forall s, r. \end{cases} \tag{7}$$

We will show experimentally that our two approximations produce reasonable results under different workload mixes and are highly competitive compared with FJ-AMVA. During our evaluation we denote the implementation of (3) with "static TP-AMVA" and (6) with "probabilistic TP-AMVA".

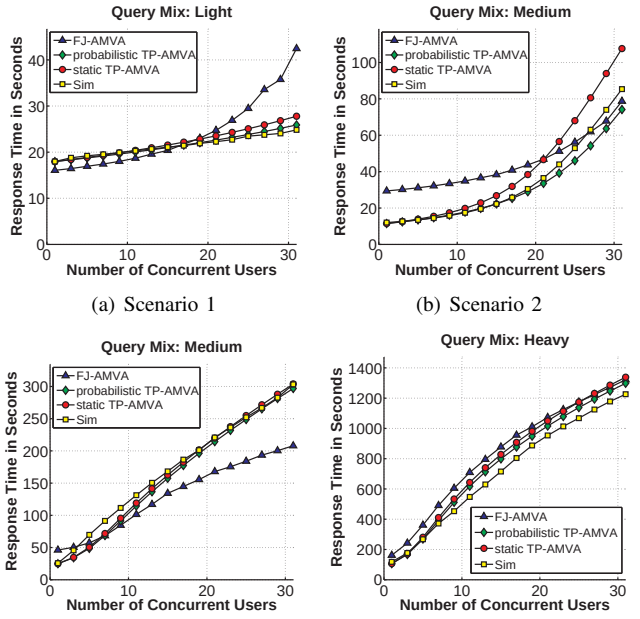
D. Evaluation

In this section we will evaluate our correction against the in-memory database simulator in [7] under different scenarios and include the FJ-AMVA into our comparison.

a) Experimental Setup: We implemented FJ-AMVA, and TP-AMVA in Matlab and conducted several experiments for different workload scenarios based on the categories: light, medium and heavy. Whereas light mixes contain mostly query classes with small degrees of parallelism and shorter execution times, heavy mixes comprise query classes with high parallelism and longer execution times. To further vary the workload, we increased the number of concurrent users from 1 to 32. Throughout all scenarios we used a fixed think time extracted from the single user scenario in our trace logs.

We used the following parameterization for the simulator, TP-AMVA and FJ-AMVA. To increase its capability of capturing resource contention more accurately, we parameterized the simulator with the fine grained query characteristics defined by b_{pr} and c_{pr}, introduced in Section III-B. For TP-AMVA we used the aggregated service demand d_r. In contrast, FJ-AMVA needs to be parameterized with the service times of jobs at each queue s_{ir}. We therefore mapped s_r^t, which naturally represents the service times needed by FJ-AMVA, onto s_{ir}. As a problem of our traces, there was no information about the placement of threads available. Hence we addressed this by applying a Monte Carlo Simulation choosing random permutations of $s_r^t = \{s_r^1, ..., s_r^t\}$ with $1 \leq t \leq J_r$ and assigning them to queue $t, 1 \leq t \leq J_r$, before running FJ-AMVA. We then took the average response time of 100 iterations, which seemed reasonable to produce stable results.

Moreover, the task scheduling system in the simulator required equal routing probabilities to each core, as does our implementation of TP-AMVA. FJ-AMVA in contrast defines its routing probabilities p_r as probability that a single queue in the fork-join construct is visited by job class r. For this case we assumed J_r/I to be a suitable approximation of p_r and thus we used $p_r = J_r/I$ to parameterize FJ-AMVA.



(a) Scenario 1 (b) Scenario 2

(c) Scenario 3 (d) Scenario 4

Fig. 5. Response Time Results for different OLAP-based Workload Scenarios

b) Results: We show the results of our experiments in Figure 5 accompanied by Table II, which depicts the mean relative response time error compared to the simulator. To give an impression of how the standard AMVA implementation (2) performs, we list its mean relative error for a run with $d_r = s_r$ and a run that takes the visit ratio into account by $d_r = l_r/I \times s_r$. As expected, AMVA clearly shows a poor overall performance. In contrast, both static and probabilistic TP-AMVA perform reasonably well throughout all scenarios and follow the trend of the simulator. Both approximations tend to be more pessimistic once the number of users increases, whereas the response time prediction under light load appears to be slightly optimistic, i.e. scenario 3. In general, our probabilistic TP-AMVA captures contention effects better than its static version, staying below a 20% error rate.

Surprisingly, FJ-AMVA lacks in its accuracy across most scenarios. Its response time prediction is too optimistic under medium mixes, i.e. scenario 3, and too pessimistic under light-medium load (scenarios 1 and 2). In most of our scenarios, we observed a very pessimistic start for FJ-AMVA, when only few concurrent users are active. This can be explained, when looking at the parallelism of our query classes. Some of our queries, such as class 1, are highly parallel with $s_r^t \approx d_r, \forall i$, and thus contain almost no synchronization time. This is why the summation over W_{ir} in FJ-AMVA, despite its scaling factor $1/i$, results in a response time that is too high. However, this effect seems to diminish when the load grows and better reflects the increasing congestion for those cases. From the results, we conclude that FJ-AMVA in its proposed form is not suitable for modeling OLAP-based query workloads, whereas our correction turns out to be reasonably accurate and due to its simplistic model a good choice for the optimization program we present in the next section.

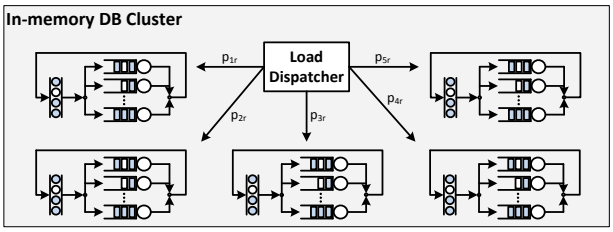


Fig. 6. Model of an in-memory Cluster Subject to Load Optimization

V. Optimization Problem

Given a parallel system with memory and resource constraints, such as an SAP HANA in-memory database cluster, we are interested in how to dispatch workloads while minimizing memory swapping. We further include utilization constraints to avoid over-provisioning in such environments by limiting the CPU utilization per server. In order to solve this constrained problem, we develop a non-linear optimization program based on our approximation probabilistic TP-AMVA, taking above mentioned constraints into account.

A. Cluster Model

We consider a simplified model of an in-memory database cluster with K servers, as depicted in Figure 6. This model contains a separate fork-join closed QN for each server, which simplifies the evaluation. However, the servers in our cluster are not completely independent, but share the same workload $\vec{N}$. We therefore define $N_{ir} = N_r \times p_{ir},\ 1 \leq i \leq K$ as the percentage of workload that goes to server i, with p_{ir} designating the probability of routing a class r request to server i.

Our optimization program aims at minimizing memory swapping, and thus minimizes the overall memory occupation M of an in-memory database cluster. To reduce the complexity of the optimization problem, we employ a less complex model for estimating M for each server by multiplying the per-class queue length with the per-class physical peak memory consumption m_r, (8b). We hereby make the pessimistic assumption that memory occupation grows as a function of the queue length Q and neglect that query classes could share data residing in main memory. Additionally, we assume that forking of new threads and joining is not related to the change of memory consumption.

B. Non-linear Optimization Problem

We minimize the memory occupation M by finding the routing probabilities p_{ir} to achieve near optimal workload placement. This results in the optimization problem given by Equation (8), with its decision variables p_{ir}, X_{ir} and W_{ir}. We integrated our probabilistic TP-AMVA in (8f) and defined $\delta_{irs} = (N_{ir} - 1)/N_{ir} \times (l_{ir}/I_i)$ for $s = r$ and $\delta_{irs} = 1$ in case of $s \neq r$. In addition, we added memory and utilization constraints in form of $M_i^{\max}$ and $U_i^{\max}$. From a performance point of view, our method uses less variables compared to FJ-AMVA, which would introduce M^2 additional binary variables to sort the response times. This gets further attention, when looking at the nature of our optimization problem, which is be non-convex. Hence, we expect the number of local optima to grow when increasing the number of classes and servers as well as introducing different constraints for each server. This exacerbates the problem of finding a globally optimal solution and requires strategies such as multi-start optimization.

$$M_{\min} = \min_{p_{ir}, X_{ir}, W_{ir}} \quad \sum_{i=1}^{K} M_i \tag{8a}$$

$$\text{s.t.:} \quad M_i = \sum_r Q_{ir}\, m_{ir},\ \forall i \tag{8b}$$

$$U_i = \sum_r X_{ir}\, \frac{l_{ir}}{I_i}\, d_{ir},\ \forall i \tag{8c}$$

$$N_{ir} = p_{ir}\, N_r,\ \forall i, r \tag{8d}$$

$$Q_{ir} = X_{ir}\, W_{ir},\ \forall i, r \tag{8e}$$

$$\sum_r Q_{ir} = \sum_r U_{ir} \left(1 + \sum_{s=1}^{R} Q_{is}\, \delta_{irs}\, \frac{l_{is}}{I}\right),\ \forall i \tag{8f}$$

$$Q_{ir} = N_{ir} - X_{ir}\, Z_{ir},\ \forall r \tag{8g}$$

$$W_{ir} \geq d_{ir},\ \forall i, r \tag{8h}$$

$$\sum_i p_{ir} = 1,\ \forall r \tag{8i}$$

$$p_{ir}, X_{ir}, W_{ir} \geq 0\ \forall i, r \tag{8j}$$

$$M_i \leq M_i^{\max},\ \forall i \tag{8k}$$

$$U_i \leq U_i^{\max},\ \forall i \tag{8l}$$

VI. Numerical Results

Our goal is to get insights in how workload placement effects the memory occupation across a cluster of in-memory databases under given constraints. We are further interested in how the performance and accuracy of multi-start based approaches for our optimization problem compare to each other. Based on empirical evidence, we show that local optimization algorithms such as interior-point are able to find good solutions under multi-start in comparison to global optimization algorithms, such as evolution strategies. In addition, we find that local optimization algorithms deliver solutions for instances up to 8 Servers and 4 classes in less than 10 minutes but lack under larger scenarios, whereas our evolution strategy handles up to 36 servers and 22 classes given the same amount of time.

A. Evaluation Scenarios

We varied the number of server instances and classes in $K, R = 2, 4, 8, 16$ and the workload N in $8K$ and $10K$ (light load) and $32K$ and $40K$ (heavy load). Our per-class populations N_r are obtained by equally dividing N across all classes, allowing fractional N_r. To investigate how R affects the total memory occupation M, we clustered our set of 22 classes with k-means (a priori normalized with z-score) across the three dimensions parallelism l_r, service demand d_r and memory occupation m_r, depicted for $R = 2, 4, 8$ in Figure 7. Finally, we set different constraints to affect the workload placement: $M_i^{\max} = 512GB,\ U_i^{\max} = 0.95$ for $i \leq K/2$ and $M_i^{\max} = 128GB,\ U_i^{\max} = 0.99$ for $i > K/2$.

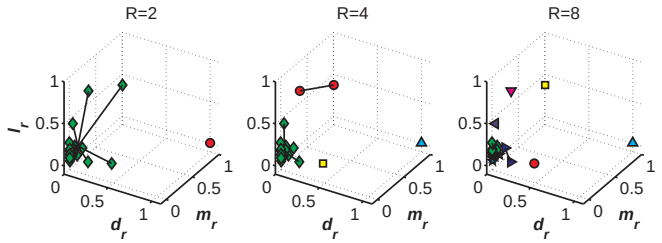


Fig. 7. Normalized k-means Clusters for different Numbers of Query Classes

B. Solution Methods

We compare the minimization of memory swapping for three different methods (local and global):

- yp: YALMIP [14] employing fmincon and the interior point algorithm
- fm: fmincon configured with the interior point algorithm
- es: $(\mu + \lambda)$ - evolution strategy [12]

Both methods fm and es, call our external solver to approximate the response time W_{ir} and throughput X_{ir}. As our methods rely on AMVA, which supports only cases with $N_{ir} \geq 1$, we applied the approximation proposed in [13], setting $Q_{ir} := 0$ when $N_{ir} < 1$. Under method yp we had to avoid the indirect division by p_{ir} in δ_{irs} and introduced an additional optimizing variable.

With method es we implemented an evolutionary algorithm called evolution strategy. In contrast to genetic algorithms, which represent solution candidates by a string of bits, evolution strategies employ a real valued encoding of solution candidates and therefore require self-adaptive mutation functions. In particular, we implemented a $(\mu + \lambda)$ evolution strategy (es), configured with a uniform parent selection and a "best selection" as environmental selection. Individuals (i.e. solution candidates) are represented by the routing probability matrix p_{ir}. On a population of $\mu = 20$ parent individuals we perform a uniform selection 100 times to produce $\lambda = 100$ mutants in each generation. In particular, we had to develop our own adaptive mutation to alter individuals obeying the constraints (8i) and (8j). To create a mutant we randomly select a server x and a class y and subsequently modify the routing probability p_{xy} by adding a new value Δp. We choose Δp uniformly from the interval $[p_{xy} - p_{xy}/\left((g \bmod 20) + 1\right), p_{xy} + (1 - p_{xy})/\left((g \bmod 20) + 1\right)]$. The amount of change introduced by this mutation in form of Δp is adapted throughout the optimization depending on the current number of generations g. Moreover, we determine the fitness of each individual by its memory occupation M. To ensure compliance with the given constraints, our implementation penalizes violations of (8k),(8l) by impairing the fitness. Finally, from the 120 individuals after mutation, the 20 fittest are selected into the next generation.

C. Evaluation Methodology

We implemented our approaches in MATLAB employing fmincon and its interior-point algorithm for yp and fm,, while es relies on an evolution strategy. Moreover, method es does not depend on MATLAB proprietary toolbox functions, and therefore could be implemented in another programming language to further decrease processing time.

TABLE IV
MEMORY OCCUPATION AND EXECUTION TIMES. TIMEOUT: 1800s†.

Instances			Memory Occupation in GB			Time in s		
K	R	N/K	fm	es	es-fS	fm	es	es-fS
2	2	8	78.9	78.9	79.4	2.3	1800.0	0.1
2	4	8	43.4	43.4	45.4	3.9	1800.0	0.1
2	8	8	23.5	23.5	24.3	20.5	1800.0	0.3
4	2	8	157.8	157.9	158.1	3.5	1800.0	0.1
4	4	8	85.7	85.7	87.5	30.0	1800.0	0.2
4	8	8	46.1	46.2	47.7	320.9	1800.0	0.4
8	2	8	315.7	315.7	315.8	21.6	1800.0	0.1
8	4	8	170.6	170.9	176.8	240.1	1800.0	0.2
8	8	8	91.6	91.6	95.3	1800.0	1800.0	0.4
2	2	32	316.4	316.4	322.5	1.5	1800.0	0.1
2	4	32	237.6	239.1	270.3	6.2	1800.0	0.3
2	8	32	121.8	123.4	146.3	31.4	1800.0	0.7
4	2	32	632.4	632.7	640.8	4.1	1800.0	0.1
4	4	32	379.6	379.8	443.0	82.1	1800.0	4.2
4	8	32	215.9	216.2	237.5	389.7	1800.0	0.7
8	2	32	1264.7	1265.2	1265.8	34.0	1800.0	2.1
8	4	32	752.5	755.6	765.9	965.5	1800.0	14.5
8	8	32	414.0	416.0	542.0	1800.0	1800.0	1.7
16	2	32	2529.3	2529.4	2531.8	576.9	1800.0	3.2
16	4	32	1504.8	1486.7	1552.0	1800.0	1800.0	51.2
16	8	32	N/A*	818.6	847.8	timeout	1800.0	25.0
16	16	32	N/A*	385.4	442.2	timeout	1800.0	3.1

†fm,es were stopped after 1800s or in case of fm when $P = 10$ solutions found
*No single solution found by fm

Our scenarios were evaluated on an Intel Core i5 CPU with 2.60GHz and two physical cores. To cope with different local optima, we randomized $P = 10$ initial points for every tuple $(K, R, N/K)$ and ran fmincon using MATLAB's MultiStart solver. Subsequently, we report the average of the cumulative execution time for all P local solver runs at a timeout of 1800 seconds to understand the performance at short time scales.

D. Results

We show the results of our optimization program for all three methods (yp,fm,es) in Table III and Table IV. We further report the cumulative execution time for yp and fm and the time until the first non-violating solution, es-fS, was found by es.

At first, we were interested in how clustering of classes affects memory optimization. In our case we observe an inversely proportional behavior of the memory occupation M once the number of classes R increases, i.e. instances (2,2,8) and (2,8,8). This can be explained by the way we calculate N_r from the fixed ratio N/K, but comparing the instances (2,2,32) and (2,8,32), where $N_r = 8$ for both cases, we see that a large difference in M remains. We also discover non-monotonicity in the overall cluster utilization U with increasing R. We show this in Table V, for a light and heavy load scenario. We impose this on classes with high parallelism and long execution times that are less often merged into a cluster with short running and sequential queries when R increases, as depicted in Figure 7. From this we conclude that the more classes are aggregated, the more inaccurate gets the estimate for M and U, given that the ratio N/K remains constant.

Another question we wanted to address is how our optimization program handles workload placement under the given constraints $M_i^{\max}$ and $U_i^{\max}$ defined in section VI-A. We therefore investigated the two instances (8,2,8) and (8,2,32) in more detail, which represent light and medium load scenarios. Table VI shows the routing probabilities p_{ir} and per-server

TABLE III
MEMORY OCCUPATION IN GB AFTER OPTIMIZATION AND EXECUTION TIMES (SECONDS). TIMEOUT: 1800S.

Instances			Memory Occupation M					U	#Success		Time		
K	R	N/K	yp	fm	es	fm-wL	es-fS	util	yp	fm	yp	fm	es-fS
2	2	10	98.785	98.766	98.766	98.766	98.769	0.06	10	10	10.854	1.531	0.088
2	4	10	55.948	54.411	54.411	54.412	54.916	0.29	10	10	6.762	5.387	0.174
2	8	10	29.976	28.740	28.740	28.740	29.137	0.28	10	10	8.950	31.181	0.266
4	2	10	197.486	197.389	197.389	197.389	197.527	0.06	10	10	4.623	2.495	0.097
4	4	10	110.972	107.603	107.546	107.603	108.631	0.29	10	10	23.365	30.590	0.196
4	8	10	59.875	56.969	56.973	56.984	57.996	0.28	10	10	33.796	310.091	0.591
8	2	10	394.973	394.734	394.777	394.777	396.131	0.06	10	10	13.243	10.741	0.123
8	4	10	221.943	215.207	215.094	215.207	224.463	0.29	8	10	120.311	361.588	0.227
8	8	10	119.734	113.816	113.831	113.940	117.097	0.28	10	8	225.114	1800.0	0.307
2	2	40	406.262	405.953	405.953	405.953	406.265	0.22	10	10	2.880	0.901	0.090
2	4	40	294.652	275.047	275.049	450.281	464.117	0.57	10	10	42.553	4.614	0.229
2	8	40	175.348	153.627	153.632	163.693	182.861	0.65	9	10	110.886	21.943	0.437
4	2	40	808.495	806.713	806.715	806.713	869.295	0.22	10	10	4.152	5.279	0.108
4	4	40	576.103	523.964	523.202	898.970	601.682	0.74	4	10	140.280	50.368	0.261
4	8	40	N/A	282.765	282.987	298.619	323.393	0.70	0	10	326.607	512.695	0.652
8	2	40	1616.150	1611.224	1613.448	1618.257	1635.765	0.22	10	10	9.793	39.926	0.804
8	4	40	N/A	1046.952	1064.830	1104.935	1125.542	0.66	0	10	321.963	557.260	1.472
8	8	40	N/A	563.991	558.313	563.991	638.667	0.78	0	2	1082.164	1800.0	1.260
16	2	40	3230.662	3221.031	3221.331	3236.513	3407.598	0.22	10	10	37.569	618.416	9.045
16	4	40	N/A	2093.377	2089.978	2144.155	2159.830	0.75	0	7	748.109	1800.0	16.054
16	8	40	N/A	N/A	1122.456	N/A	1159.027	0.74	0	0	timeout	timeout	47.863
16	16	40	N/A	N/A	483.828	N/A	646.131	0.55	0	0	timeout	timeout	0.797

memory occupation M_i for the two solutions found by fm. Under light load we see that fm tries to achieve as little interference as possible between the two classes, resulting in $\max_i(M_i) = 52.4GB$. Moreover, as no constraints are violated, the placement can be an arbitrary permutation across i, but needs to remain fixed for r. Once the workload grows to $N_{ir} = 128$, which is a normal scenario for SAP HANA dealing with more than 128 parallel connections, the memory constraints for server 5-8 are violated. At this point we observe a workload shift towards servers 1-4, occupying 250.9GB on each. This suggests that our approach is able to optimize constrained workload placement reasonably well.

All of our methods (yp,fm,es) produce similar results regarding M for instances where all P solver runs completed successfully, i.e. (4,2,10). We explain this due to the same algorithm that is used to solve the queueing models. Though theoretically identical yp applies the interior-point algorithm, whereas fm and es depend on a fixed-point iteration, on which we impose the slightly different results between yp and fm. In contrast, we expect a higher computational cost for fm due to the external solver call. However, during our experiments we observed that despite slightly more efficient code produced by YALMIP, yp needed more iterations to converge than fm at same tolerances levels for the stopping criteria. Looking at the execution times, we see that yp and fm are able to find solutions in less than 180 seconds for 4 servers and 4 classes, if not necessarily for all P initial conditions. Our Evolution Strategy seems more efficient, as it reports the first solution after 47.9 seconds in scenarios such as (16,8,40), where yp and fm were timed out before a single successful solution was found. We explain this due to the implementation of es, which involves less evaluations of the objective function per iteration. Due to the increasing number of decision variables in larger scenarios, yp required up to 400MB of main memory under (16,8,40). By contrast, our external solver based methods fm and es require only a few megabytes. Concluding the results,

TABLE V
AVERAGE SERVER UTILIZATION FOR 8-SERVER SCENARIO (K=8)

N/K=8			N/K=32		
R=2	R=4	R=8	R=2	R=4	R=8
0.05	0.24	0.22	0.18	0.76	0.71

we showed that both techniques interior point based methods and evolution strategies can be effectively used to solve our constrained memory optimization problems.

VII. RELATED WORK

Research into in-memory database performance started in 2002 when [16] introduced fundamental cost models including the entire memory hierarchy in a database system. Nowadays, on-demand provisioning of these systems drives research further into database optimization employing QNs [17].

In [18] classification-based machine learning is used to schedule tenants in multi-tenant databases. The authors characterize tenant and node-level behavior based on performance metrics collected from database and OS layer and validate their framework in a PostgreSQL environment. However, in this work scheduling constraints are only approximated. Workload characterization and response time prediction via non-linear regression techniques for in-memory databases are proposed in [19]. The authors derive tenant placement decisions by employing first fit decreasing scheduling, but evaluate on small scale only. [20] propose a new framework for managing performance SLOs under multi-tenancy scenarios. Their work combines mathematical optimization and boolean functions to enable what-if analyses regarding SLOs, but relies on brute force solvers and ignores OLAP workloads. In [21] query demands are quantified by a fine-grained CPU sharing model including largest deficit first policies and a deficit-based version of round robin scheduling. The methodology applies to database-as-a-service platforms and is validated on a prototype of Microsoft SQL Azure. This work neglects characteristics for memory occupation. [22], [23] introduce frameworks for non-linear cost optimization regarding SLA violations and

TABLE VI
WORKLOAD PLACEMENT UNDER SCENARIOS (8,2,8) AND (8,2,32)

N/K		*i*=1	*i*=2	*i*=3	*i*=4	*i*=5	*i*=6	*i*=7	*i*=8
7	p_{i1}	0	0.167	0.167	0	0.167	0.167	0.167	0.167
7	p_{i2}	0.500	0	0	0.500	0	0	0	0
7	M_i	0.6GB	52.4GB	52.4GB	0.6GB	52.4GB	52.4GB	52.4GB	52.4GB
28	p_{i1}	0.200	0.200	0.200	0.200	0.100	0	0.100	0
28	p_{i2}	0	0	0	0	0	0.5	0	0.5
28	M_i	250.9GB	250.9GB	250.9GB	250.9GB	128.0GB	2.5GB	128.0GB	2.5GB

resource usage, applied to web service based applications and cloud databases. Their work either relies purely on constraint definitions or does not consider closed QNs. [24] proposes a framework for multi-objective optimization of power and performance. The methodology applies to software-as-a-service applications and it is validated using a commercial software, SAP ERP. The approach is based on simulation and does not consider thread-level fork-join.

[25]–[27] use multi-variate regression and analytical models of closed QNs to predict query performance based on logical I/O interference in multi-tenant databases. However, these methods require detailed query access pattern and are evaluated for small numbers of jobs and batch workloads only. Ignored by the latter, thread-level fork join is addressed by [7] and [6], but despite using similar techniques, their approaches are either computationally expensive or rely on exponential service time distributions.

VIII. CONCLUSIONS AND FUTURE WORK

In this paper we have made several contributions, which include a novel analytic response time approximation that models thread-level fork join and per-class memory occupation in in-memory systems. In addition, we have developed an optimization based formulation that facilitates our analytic approximation and efficiently seeks for load dispatching routing probabilities to minimize memory swapping in in-memory database clusters. Furthermore, we have shown that our models exceed the accuracy of existing approaches using real traces from a commercial in-memory database appliance, SAP HANA, for validation.

Directions for future work include a more sophisticated model to estimate memory consumption as well as a proof of concept for our models through thorough experimentation and validation with different hardware configurations. In addition, we plan to implement our provisioning framework in a real in-memory database system and extend it by new features to include shared memory access and multi-tenancy.

REFERENCES

[1] "SAP HANA Performance: Efficient Speed and Scale-Out for Real-Time Business Intelligence," 2013. [Online]. Available: http://www.sap.com/bin/sapcom/downloadasset.sap-hana-performance-pdf.html

[2] J. Schaffner, B. Eckart, C. Schwarz, J. Brunnert, D. Jacobs, and A. Zeier, "Towards Analytics-as-a-Service Using an In-Memory Column Database," in *New Frontiers in Information and Software as Services SE - 11*, ser. Lecture Notes in Business Information Processing, D. Agrawal, K. Candan, and W.-S. Li, Eds. Springer Berlin Heidelberg, 2011, vol. 74, pp. 257–282.

[3] P. J. Schweitzer, "Approximate analysis of multiclass closed networks of queues," in *Proc. of the Int'l Conf. on Stoch. Control and Optim.*, Amsterdam, 1979, pp. 25–29.

[4] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "SAP HANA database: data management for modern business applications," *SIGMOD Rec.*, vol. 40, no. 4, pp. 45–51, Jan. 2011.

[5] J. Rolia, G. Casale, D. Krishnamurthy, S. Dawson, and S. Kraft, "Predictive Modelling of SAP ERP Applications: Challenges and Solutions," in *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, ser. VALUETOOLS '09, 2009, pp. 9:1—-9:9.

[6] F. Alomari and D. Menascé, "Efficient Response Time Approximations for Multiclass Fork and Join Queues in Open and Closed Queuing Networks," p. 1, 2013.

[7] S. Kraft, G. Casale, A. Jula, P. Kilpatrick, and D. Greer, "WIQ: Work-Intensive Query Scheduling for In-Memory Database Systems," in *Proc. of IEEE CLOUD*, 2012.

[8] A. Jula, "Multicore scalability and NUMA effects on HDB with SAP-H data and TPC- H queries," SAP, Tech. Rep., 2012.

[9] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios, "Open, Closed, and Mixed Networks of Queues with Different Classes of Customers," *Journal of the ACM*, vol. 22, no. 2, pp. 248–260, 1975.

[10] E. Lazowska, J. Zahorjan, G. Graham, and K. Sevcik, *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice Hall, 1984.

[11] E. Varki, "Response time analysis of parallel computer and storage systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 11, pp. 1146–1161, 2001.

[12] H.-G. Beyer and H.-P. Schwefel, "Evolution strategies – A comprehensive introduction," *Natural Computing*, vol. 1, no. 1, pp. 3 – 52, 2002.

[13] R. Suri and R. Shinde, "Using Fractional Numbers of Customers to Achieve Throughputs in Queueing Networks," in *Proceedings of the Industrial Engineering Research Conference (IERC)*, 2005.

[14] J. Loefberg, "Yalmip : A toolbox for modeling and optimization in MATLAB," in *Proc. of CACSD*, 2004.

[15] K. Molka and G. Casale, "Memory-Aware Sizing for In-Memory Databases," Dept. of Computing, Imperial College London, Tech. Rep., 2014. The technical report can be made available to the reviewers upon request to the chair.

[16] S. Manegold, P. Boncz, and M. L. Kersten, "Generic database cost models for hierarchical memory systems," in *Proceedings of the 28th international conference on Very Large Data Bases*, ser. VLDB '02, 2002, pp. 191–202.

[17] R. Osman and W. J. Knottenbelt, "Database System Performance Evaluation Models: A Survey," *Performance Evaluation*, vol. 69, no. 10, pp. 471–493, 2012.

[18] A. J. Elmore, S. Das, A. Pucher, D. Agrawal, A. El Abbadi, and X. Yan, "Characterizing tenant behavior for placement and crisis mitigation in multitenant DBMSs," in *Proceedings of the 2013 International Conference on Management of Data*, ser. SIGMOD '13, 2013, pp. 517–528.

[19] J. Schaffner, B. Eckart, D. Jacobs, C. Schwarz, H. Plattner, and A. Zeier, "Predicting In-Memory Database Performance for Automating Cluster Management Tasks," in *2011 IEEE 27th International Conference on Data Engineering*, 2011, pp. 1264–1275.

[20] W. Lang, S. Shankar, J. M. Patel, and A. Kalhan, "Towards Multi-tenant Performance SLOs," in *2012 IEEE 28th International Conference on Data Engineering*, 2012, pp. 702–713.

[21] S. Das, V. R. Narasayya, F. Li, and M. Syamala, "CPU Sharing Techniques for Performance Isolation in Multitenant Relational Database-as-a-Service," in *VLDB*, 2013, pp. 37–48.

[22] J. Almeida, V. Almeida, D. Ardagna, C. Francalanci, and M. Trubian, "Resource Management in the Autonomic Service-Oriented Architecture," in *2006 IEEE International Conference on Autonomic Computing*, 2006, pp. 84–92.

[23] J. Rogers, O. Papaemmanouil, and U. Cetintemel, "A generic auto-provisioning framework for cloud databases," in *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW)*, 2010.

[24] H. Li, G. Casale, and T. Ellahi, "SLA-driven planning and optimization of enterprise applications," in *Proceedings of the first joint WOSP/SIPEW*, 2010.

[25] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal, "Perfor-

mance prediction for concurrent database workloads," in *Proceedings of the 2011 international conference on Management of data*, ser. SIGMOD '11, 2011, pp. 337–348.

[26] W. Wu, Y. Chi, H. Hacigümüs, and J. F. Naughton, "Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads," *PVLDB*, vol. 6, no. 10, pp. 925–936, 2013.

[27] R. Suri, S. Sahu, and M. Vernon, "Approximate Mean Value Analysis for Closed Queuing Networks with Multiple-Server Stations," in *Proc. of the 2007 Industrial Engineering Research Conference*, 2007.