Contention-Aware Workload Placement for In-Memory Databases in Cloud Environments

Karsten Molka, Imperial College London, SAP HANA Cloud Computing, Systems Engineering
Giuliano Casale, Imperial College London

Big data processing is driven by new types of in-memory database systems. In this paper we apply performance modeling to efficiently optimize workload placement for such systems. In particular, we propose novel response time approximations for in-memory databases based on fork-join queuing models and contention probabilities to model variable threading levels and per-class memory occupation under analytical workloads. We combine these approximations with a non-linear optimization methodology that seeks for optimal load dispatching probabilities in order to minimize memory swapping and resource utilization. We compare our approach with state-of-the-art response time approximations using real data from an SAP HANA in-memory system and show that our models markedly improve accuracy over existing approaches, at similar computational costs.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: *Modeling techniques*; G.1 [**Numerical Analysis**]: Optimization—*Constrained optimization, Nonlinear programming*; G.3 [**Probability and Statistics**]: *Queueing Theory*

General Terms: Performance, Management

Additional Key Words and Phrases: In-memory Databases, Variable Threading Levels

1. INTRODUCTION

In-memory database systems are a key driver for big data analytics. In addition to processing data solely in main memory, these systems exploit latest hardware technologies, including flash storage, FPGAs and GPUs, to sharply optimize request 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 [SAP 2013]. As a result, in-memory systems are in high commercial demand as part of cloud software-as-a-service offerings [Schaffner et al. 2011b]. This poses new challenges to 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.

The work of Giuliano Casale is supported by the EU project MODAClouds FP7-318484.
Author's address: K. Molka, The Concourse, Queen's Road, Belfast BT3 9DT, UK; and 180 Queen's Gate, London, SW7 2AZ, UK; email: k.molka13@imperial.ac.uk; Giuliano Casale, Imperial College London, 180 Queen's Gate, London, SW7 2AZ, UK; email: g.casale@imperial.ac.uk

DOI:http://dx.doi.org/10.1145/0000000.0000000

Providing support for workload placement decisions is one of these challenges. To solve this challenge we require optimization models that are able to capture in-memory database characteristics, such as variable threading levels and memory occupation. Research is increasingly focusing on management problems of this kind. In particular, recent work on consolidation and scheduling of applications in cloud environments emphasizes the importance of accounting for different resource and workload dimensions in order to find good solutions to provisioning problems, [Zhang et al. 2014; Mastroianni et al. 2013]. Several other works address the challenges of predicting workload performance using machine learning techniques, buffer pool and queueing models [Duggan et al. 2014; Wu et al. 2013; Elmore et al. 2013; Schaffner et al. 2011a], but lack in accounting for the highly-variable threading levels of analytical workloads when processed by in-memory databases.

As a consequence, this paper introduces a novel load dispatching framework to address decision support challenges in both planning and operational phases. It tackles the problem of placing analytical workloads on in-memory database clusters, which provide back-ends for cloud-based services. In particular, our framework seeks for workload routing probabilities that can load balance instances of in-memory databases. Thus it can be directly used inside an in-memory database management framework. Alternatively, our framework can be used to guide what-if analyses that aim at exploring the effects of different hardware system configurations on performance and total cost of ownership.

The contribution of our framework is twofold. First, we introduce a novel queueing predictive model to describe the levels of contention at in-memory database resources. This allows us to establish the likelihood that an in-memory database configuration will comply to service level objectives (SLOs) in place with customers. Second, our framework integrates this predictive model with an optimization-based formulation in order to seek for optimal load dispatching probabilities. Moreover, since memory exhaustion and swapping are likelier to happen with memory-intensive analytical workloads, it is crucial that both performance and optimization models are able to capture memory constraints. Conversely, existing sizing methods for enterprise applications have primarily focused on modeling mean CPU demand and request response times. One reason for this appears to be the difficulty in modeling memory occupation. This particularly requires the ability to predict the probability of a certain mix of queries being active at any given time. However, probabilistic models tend to be expensive to evaluate, 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 [Schweitzer 1979]. We observe in particular that current AMVA methods are unable to correctly capture the effects of variable threading levels inside an in-memory database system. Consequently, we propose a correction, called thread-placement AMVA (TP-AMVA), that markedly improves accuracy. Our approach retains the same computational properties of AMVA and is simple and inexpensive to integrate into optimization programs. We also demonstrate that multi-start interior point methods can be effectively used to solve our workload placement problem. In addition, we provide and extensive evaluation of our approach using real traces from a commercial in-memory database, SAP HANA [Färber et al. 2011].

This paper is an extension of the conference paper in [Molka et al. 2014]. A new response time approximation is here proposed that introduces load dependent contention probabilities, improving the accuracy of predictions significantly. Moreover, we introduce a generic optimization methodology and compare it against global optimization. Furthermore, we propose a refinement step to our optimization methodology and

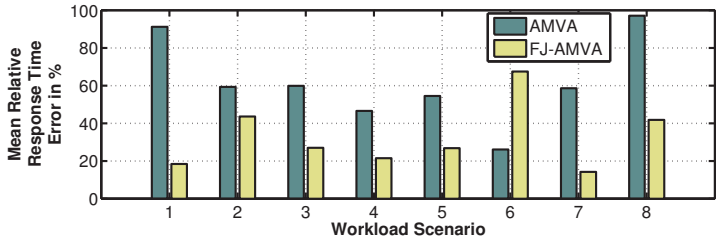


Fig. 1. Predicted Response Time Error vs. Simulation

validate expected improvements against simulation. Summarizing, our main contributions are:

— A novel analytic response time approximation for in-memory databases that considers thread-level fork-join and contention probabilities
— An extensible optimization methodology that seeks load-dispatching probabilities in order to optimize performance and cost for in-memory clusters subject to resource constraints
— An experimental validation that confirms applicability of our optimization method for large-scale models with hundreds of servers using trace-driven model parametrization based on performance measurements from a commercial in-memory database

The remainder of this paper is organized as follows. Section 2 motivates our research objective. Section 3 introduces the characteristics of our in-memory database system and presents a novel response time approximation, which is evaluated against real traces from a commercial in-memory database in Section 4. In Section 5 we develop an extensible sizing methodology based on our response time approximation and give a numerical evaluation in Section 6. Section 7 discusses related work, whereas Section 8 concludes this paper and outlines future work.

2. MOTIVATION

In-memory databases are an important type of big data analysis systems capable of processing memory-intensive workloads in a parallel fashion. In order to support sizing decisions for such systems it is essential to develop models that are able to capture the key properties of in-memory databases, such as response times and request throughputs. However, existing analytical approaches, such as AMVA [Schweitzer 1979], widely used to model the performance of multi-tier applications [Rolia et al. 2009], and state-of-the-art AMVA based methods, i.e. fork-join AMVA (FJ-AMVA) [Alomari and Menascé 2014], lack in correctly capturing the extensive and variable threading-level introduced by analytical workloads. To demonstrate this, we parameterized these two methods from real traces of an in-memory database SAP HANA and compared their response time predictions with a validated in-memory database simulator [Kraft et al. 2012]. We give an excerpt of these results in Figure 1, which depicts the relative response time error of AMVA and FJ-AMVA compared with our simulator under different workloads. We observe that using both AMVA and FJ-AMVA can occasionally result in large prediction errors. In particular, we found that our traces do not meet the exponentiality assumptions and thus the assumptions of FJ-AMVA, which is one of the reasons for its performance on our dataset. Summarizing, our results clearly motivate the need for enhanced in-memory database performance models that can cope with extensive variable threading-levels introduced by analytical workloads.

Secondly, we are interested into the peak memory occupation of an in-memory database cluster under particular workload placements. More specifically, we infer the

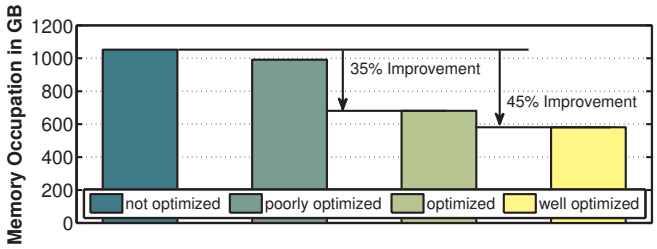


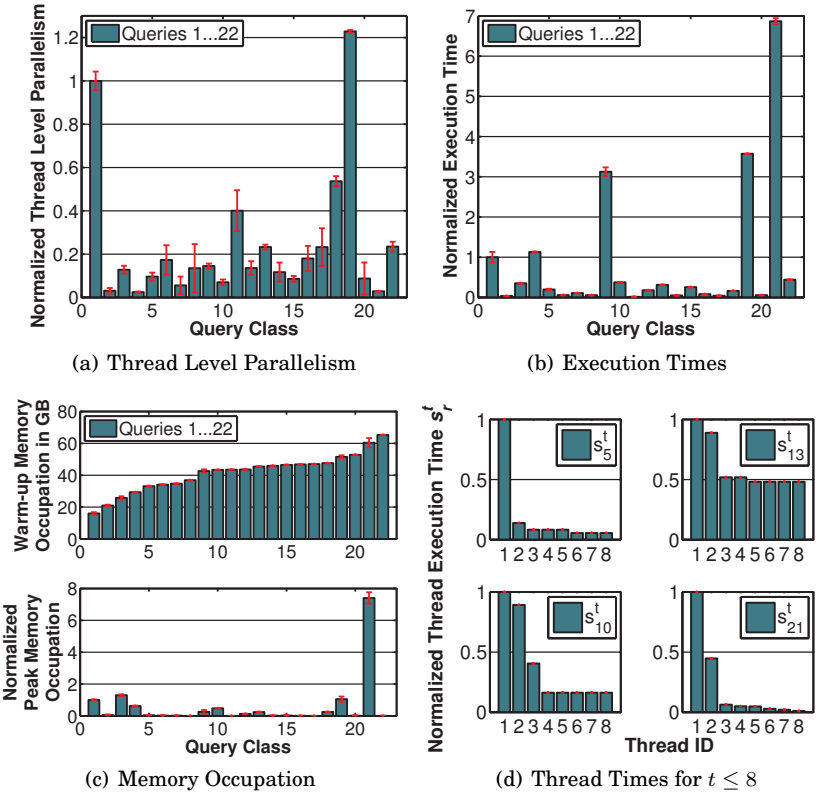
Fig. 2. Potential Improvement of Resource Usage

memory occupation from the number of jobs that are concurrently processed in such a cluster, as detailed in Section 5. To do so, we have integrated our novel response time approximation TP-AMVA into an optimization model and computed the respective number of jobs that are in contention for resources at each server. The solution of this optimization model is a workload placement, which impacts the memory occupation of our cluster. Consequently, we looked at four different workload placements in a four server scenario and analyzed the resulting memory occupation. In particular, we considered the following workload placements: *not optimized* (distributing workload equally), *poorly optimized* (worst solution found by a local solver), *optimized* (best solution found by a local solver) and *well optimized* (best solution found by our optimization refinement step). As we reveal in Figure 2, workload placement has a huge impact on the memory occupation, indicating that improvements of memory usage up to 45% are possible compared to a non-optimized workload placement. This strongly motivates our approach of efficiently seeking for optimal workload placements.

3. MODELING IN-MEMORY DATABASE PERFORMANCE

3.1. Database Characteristics under OLAP

In-memory database systems provide backends to on-premise enterprise applications and on-demand cloud-based services. In particular, in-memory databases are optimized to execute analytical business transactions, *i.e.*, OLAP. These types of transactions 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. Before going into detail about the modeling of such in-memory database systems we will discuss their diverse characteristics under OLAP workloads first. For this we analyzed trace logs from benchmark experiments running SAP HANA on an IBM X5 4-socket database server configured with 1TB main memory, collected in [Jula 2012]. The benchmark was run at 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 3(a)-(c). All values have been obtained from isolated query runs and are shown with their respective standard deviations. For confidentiality, we have normalized our results by the respective value of class 1. In Figure 3(a) we present the average number of CPU cores used by each query class and denote this with thread level parallelism l. As expected, we see a strong variability of the parallelism across all query classes, which can increase contention for resources under OLAP workload mixes. In addition, we observe a varying computational expense for all OLAP queries, depicted in Figure 3(b). We further reveal the memory intensive character of OLAP workloads in Figure 3(c) by showing the peak physical memory temporarily occupied during the processing of queries, which varies on gigabyte scale. To emphasize the importance of compres-



(a) Thread Level Parallelism (b) Execution Times

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

Fig. 3. OLAP Workload Characteristics (normalized by respective value of Query Class 1)

sion during the execution of OLAP workloads, we demonstrate in Figure 3(c) that our benchmark dataset with a size of 1.3 TB was reduced to approximately 65GB after conducting a warm-up run for each query class to pre-load required data into main memory.

3.2. In-memory Database Server Model

Since our in-memory system is intensively used for business analytics, similar types of requests coming from analytics applications will recurrently hit the database system. The TPC-H benchmark used for our experiments simulates this behavior of a fixed set of users that recurrently submit their requests to the database. Hence, this suggests to use a closed workload model.

The execution of requests submitted by our benchmark involves to major stages, query planning and execution. From a high level point of view, the planning phase involves the analysis of query structures by a query planner that subsequently creates an appropriate job execution plan. During the execution phase job execution plans are forwarded to an admission buffer and depending on the query plan parallelism processed by one or several worker threads, whereby each worker thread is assigned to an available CPU core. Before worker threads can complete their task, processed information has to be synchronized, e.g. parallel data aggregation, before a query can leave the system.

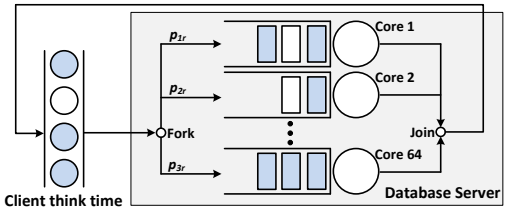


Fig. 4. Multiclass Fork-Join Queueing Model of a Database Server

In order to model the query execution, performance models for in-memory databases require a contention model that accurately captures hardware properties and application characteristics as introduced by analytical workloads. Hence and motivated by the high level of query parallelism shown in Figure 3(a), we apply fork-join queues to model the execution of worker threads on processing cores of a multi-core in-memory database system. In particular, we consider processor sharing (PS) queues, where service times are generally distributed i.i.d. random variables [Baskett et al. 1975], and combine these with a multiclass closed queueing network. This allows us to model the execution of different workload classes that are recurrently submitted by a fixed set of users, as it is the case for the TPC-H benchmark. To model this behavior more accurately, we additionally employ a think time model that captures the time between two request submissions. In addition to this, the think time model accounts for database internal scheduling mechanisms. These mechanisms rely in particular on admission buffers, an important part of the complex query processing and scheduling engines of in-memory database systems, used to delay job executions in case database internal resources, such as thread pools, are exhausted. In Figure 4 we show our queueing model used to represent an in-memory database server. This queueing model captures the behavior of query jobs split into several tasks on arrival at the system, which are then processed by worker threads and 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 queueing networks (QNs) via simulation, e.g. in [Kraft et al. 2012], 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 and give the relevant notation in Table I.

3.3. Approximations to Fork-Join Queues

The widely used exact analytical solution for closed QNs, known as mean-value analysis (MVA), determines the response time W_{ir} for a job of class r at queueing center (core) i depending on the total number of per-class jobs $\vec{N}$ in a system as follows [Lazowska et al. 1984]:

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

Here, the response time is estimated by the service demand d_{ir} of the arriving job r at core i inflated by the number of jobs already queueing at i. More specifically, d_{ir} can be expressed as $v_{ir}s_{ir}$, the product of visits v_{ir} to queue i and the service time s_{ir} at queue i, required in cases where a job is routed back to a queue before arriving at the join station. Furthermore, the arrival instant queue $A_{ir}(\vec{N})$ counts for the total number of

Table I. Main Notation

Symbol	Description
Workload Parameters	
R	Number of query classes
b_p	Length of processing phase p
c_p	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
s_r^t	Service time of thread t of class r
T_r	Number of threads per class r
$\vec{N}$	Vector with number of per-class jobs: $N_1, ..., N_R$
$\vec{Z}$	Vector of per-class think times $Z_1, ..., Z_R$
Additional Parameters	
I_i	Number of available processing cores at server i
p_{ir}	Probability of class r jobs being routed to station 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

jobs queuing or being served 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 represents the queue length with one class r job less. MVA is applied in a recursive fashion, but gets intractable for problems with more than a few customer classes. This is addressed by Bard-Schweitzer [Schweitzer 1979], proposing an approximate MVA (AMVA) that employs a fixed-point iteration and estimates A_{ir} via linear interpolation:

$$A_{ir}(\vec{N}) \approx \sum_{s=1}^{R} Q_{is}\, \delta_{rs}(\vec{N}), \tag{2}$$

$$\delta_{rs} = \begin{cases} \dfrac{N_r - 1}{N_r}, & s = r \\ 1, & s \neq r, \quad \forall s, r. \end{cases} \tag{3}$$

However, temporal delays introduced by synchronization in fork-join queues cannot be described with the above product-form models. Since MVA and AMVA are not applicable in that case, more recent approaches have tried to address this aspect [Varki 2001; Alomari and Menascé 2014]. In particular, [Alomari and Menascé 2014] propose a response time approximation called FJ-AMVA that sorts per-class residence times in descending order and scales them by a coefficient based on harmonic numbers for better estimation of the synchronization overhead. Both approaches assume 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, $\max_i(s_{ir}) \times H_{T_r}$ equals $s_r^* = E[\max_i(\tilde{S_{ir}})]$, where s_r^* becomes the maximum service time of a job and $H_{T_r} = \sum_{t=1}^{T_r} t^{-1}$ denotes the t-th harmonic number for job class r with T parallel tasks. While FJ-AMVA treats the heterogeneous case, in which s_{ir} does not have to be the same at every queue, differently than the authors in [Varki 2001], both fork-join approximations require exponentially distributed service times. However, in our traces we observed that service times for all 22 TPC-H queries do not show an exponential distribution, but instead a generally low variability. We point this out in Figure 3(b)

and 3(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. In our case, a maximum variability of ≈10% occurs for the TPC-H query template Q1. We think that relying on harmonic numbers is not a favorable approach for scenarios with no exponentialty in service demands as it is the case for our traces. Hence, we expect this low variability to be problematic for FJ-AMVA, which motivates the need for a response time correction that does not rely on exponential service times.

3.4. Response Time Correction

Since thread-level fork-join cannot be directly expressed with (1) we propose an analytical response time correction called TP-AMVA, which considers the placement of tasks in fork-join queues and unlike FJ-AMVA does not rely on exponential service time distributions. In particular, we approximate the fork-join construct with only one single queue, which decreases processing time and simplifies its integration into our optimization program. This abstraction does not consider the state of individual queues, but rather the average state of the system, which follows the MVA paradigm. Since we assume queues to be all with the same processing rates and equal class routing probabilities, their mean queue length will be the same. Thus, if we want to enforce SLAs, it is sufficient to consider the expression of just a single arbitrary queue. Moreover, since we consider jobs to not cycle within the fork-join construct, $d_r = v_r s_r = s_r$.

In the following we will give an incremental approach that is helpful to understand how each additional extension to the AMVA expression contributes to accuracy.

3.4.1. Thread-level Parallelism. At first, we introduce the query thread level parallelism l into the MVA expression in (1), since this is an important workload property, largely ignored in previous work. Our correction has the following form:

$$W_r = d_r \left(1 + \sum_{s=1}^{R} Q_s \, \delta_{rs} \, \frac{l_s}{I} \right), \tag{4}$$

where the response time W_r is calculated as the service demand d_r inflated by a factor that describes the service rate degradation under processor sharing due to jobs, which already compete for resources at the same queue. This factor is represented by the arrival queue length $A_s = Q_s \, \delta_{rs}$, which we estimated by employing the Bard-Schweitzer approximation. We then correct A_s by the factor l_s/I to estimate the per-core queue length in a system with I cores based on the query parallelism l. This is possible, because we record thread-level information for each query class allowing us to better approximate the fork-join feature. Response times W_r, throughputs X_r and queue lengths Q_r can then be obtained by performing the AMVA fixed-point iteration. Similarly to the arrival queue length, we approximate the utilization in a fork-join system as

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

Considering our assumptions about same processing rates and equal routing probabilities, it is sufficient to take the expression of an individual arbitrary queue to obtain the mean total system utilization.

3.4.2. Static Contention Probabilities. We now improve the expression in (4) further by an empirical calibration that considers static contention probabilities. This second step follows the idea that an arriving class r job affects W_r and Q_r depending on its routing probability p_r to a particular queue in the fork-join construct. We account for this effect

in the second part of the summation term, by multiplying the class r queue length Q_r with p_r, rather than scaling d_r, since we have to guarantee that job r sojourns for at least d_r in the system. This refinement step results in the following expression:

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

where p_{rs} is defined as:

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

While (6) retains the same computational properties of (4), we expect (6) to result in a more accurate estimation of response times under concurrent workloads.

3.4.3. Load-Dependent Contention Probabilities. In this final step, we further improve the definition of contention probabilities over (7). This extension modifies the queue length based on the probability of query pairs interfering with each other depending on the server utilization. With such an approach we expect to be able to distinguish the impact of contention effects under light and heavy load scenarios more accurately. We therefore define p_{rs} as

$$p_{rs} = U + (1 - U) \, \frac{l_r}{I} \, \frac{l_s}{I}, \quad \forall s, r. \tag{8}$$

The idea behind this approach is twofold. Under light load the first summand in (8) can be neglected, since the system utilization is at a low level. That means the major contribution comes from the term $(l_r/I) \times (l_s/I)$, expressing the probability that queries of class r are placed on the same queue as queries of class s. Under heavy load we set this probability to one, since we assume that if the number of parallel users is large enough it will be unlikely that two queries do not interfere with each other. This is expressed by the first summand in (8), which becomes 1.0 while the contribution of the second summand goes against zero. While we expect (8) to markedly improve accuracy over (4) and (6), it introduces a higher level of complexity than the latter when used in combination with nonlinear optimization. Hence, with our three AMVA extensions we are facing the common problem of choosing the right tradeoff between suitability of mathematical models for nonlinear optimization and their accuracy/complexity for respective predictions. To better justify which of the three AMVA extensions is most suitable for our optimization problem, we will first give an extensive experimental evaluation in the next Section. During our evaluation we denote the implementation of (4) with TP-AMVA$_{\text{stat}}$, (6) with TP-AMVA$_{\text{prob}}$ and (8) with TP-AMVA$_{\text{prob util}}$.

4. PREDICTION MODEL VALIDATION

4.1. Experimental Setup and Methodology

To understand the performance of our queueing predictive models, we will validate its per-class prediction accuracy against real traces from an IBM 4-socket in-memory database system. Subsequently, we will conduct a sensitivity analysis to explore the robustness of our technique under concurrent workloads while increasing the number of processing cores. We also refer to the validation against an in-memory database simulator in our previous work [Molka et al. 2014].

4.1.1. Database Server Configuration and Trace Logs. For our evaluation we consider the TPC-H benchmark traces introduced in Section 3. These traces record measurements from isolated runs for all 22 TPC-H query templates as well as response times,

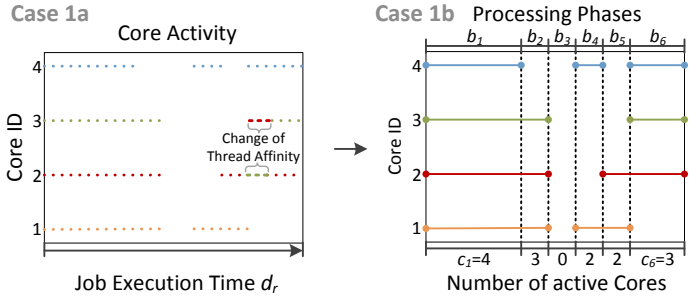


Fig. 5. Service Demand Estimation for an OLAP Query

throughputs and inter arrival times for benchmark scenarios with 1,4,8,16 and 32 concurrent users. We will use the former to parameterize our models, whereas the latter will be considered for evaluation of the model prediction accuracy under concurrent workloads. In particular, we consider the traces for three different hardware systems, each with the same SAP HANA installation: an IBM 4-socket system (IBM4) with 1TB of main memory as well as the two 8-socket systems IBM8 and HP8, both configured with 2TB main memory. For each of these systems, 2-socket and 4-socket NUMA (non-uniform memory access) configurations were benchmarked, including the 8-socket configuration under IBM8 and HP8. To account for the different system parameters under these additional configurations, such as the varying number of processing cores and service times, we have run our IBM4 trace log analysis described in Section 3 on the available datasets from the new 2-socket, 4-socket and 8-socket NUMA configurations.

4.1.2. Service Demand Estimation. To parameterize our queueing model presented in Section 3.2, we need to extract per-class service times and parallelism from the available traces. Since [Kraft et al. 2012] extracted theses parameters to drive their in-memory database simulator, we review their process briefly and subsequently extend it for use with our analytical model. To better illustrate this process, we represent our traces in Figure 5 by an exemplary job that is executed on a 4-core system. Figure 5 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 [Kraft et al. 2012], with the aim to provide fine-grained service requirements. However, for our approximation we favor a less complex parameterization that avoids additional processing overhead when integrated into optimization programs. This is why we determine the per-class service time d_r and thread-level parallelism l_r for use with our analytical model as aggregates of these measurements, $d_r = \sum_{p=1}^{P} b_{pr}$ and $l_r = {d_r}^{-1} \sum_p b_{pr} c_{pr}$. Since the parameterization of FJ-AMVA is similar, but relies on execution times of tasks pertaining to a query process, we refer to Appendix A for a more detailed description.

4.1.3. Model Parameterization. To conduct the prediction model evaluation, we implemented AMVA, FJ-AMVA, and TP-AMVA in MATLAB R2014a and used the following

parameterization based on our estimated per-class service times and thread-level information.

For AMVA and TP-AMVA we used the aggregated service demand d_r, where jobs visit processing queues only once. We also included an alternative parameterization of AMVA with $d_r = (l_r/I)s_r$ to explore its accuracy when using service times scaled by the thread level parallelism over the number of available processing cores. Throughout the evaluation we denote this parameterization with $\text{AMVA}_{\text{visits}}$. In contrast, FJ-AMVA needed to be parameterized with the service times of jobs at each queue s_{ir}. As detailed in Appendix A, we obtained these values from execution times of each active worker thread s_r^t running during execution of a class r job. We then mapped s_r^t, which naturally represent the service times needed by FJ-AMVA, onto s_{ir}, where t is limited by the maximum number of threads T_r per class r. As a problem of our traces, the available information about the placement of threads was insufficient. Hence we addressed this by applying a Monte Carlo Simulation choosing random permutations of $s_r^t = \{s_r^1, ..., s_r^t\}$ with $1 \leq t \leq T_r$ and assigning them to queue t, $1 \leq t \leq T_r$, before running FJ-AMVA. We then took the average response time of 100 iterations, which seemed reasonable to produce stable results. Finally we approximated the class routing probabilities p_r, with $p_r = 1/l_r$ for our TP-AMVA implementation and $p_r = T_r/I$ for FJ-AMVA.

4.2. Prediction of TPC-H Query Templates

4.2.1. Prediction Scenarios and Methodology. At first, we were interested in the per-class prediction accuracy of TP-AMVA under different multi-programming levels, including 1,4,8,16 and 32 concurrent users (Con). We parameterized AMVA, FJ-AMVA and TP-AMVA with system parameters of the IBM4 system, obtained from isolated query runs. Subsequently, we predicted the per-class response time for each of the $R = 22$ TPC-H query templates under concurrent workloads. Since each workload scenario is defined by a class population vector $\vec{N} = N_1, ..., N_R$ and a think time vector $\vec{Z}$, we used the respective trace think times for each concurrent user scenario (Con_i) and defined the population for class r as $N_r = \text{Con}_i$.

Due to the amount of workload scenarios across all prediction methods and query templates, we only give the trend of the per-class prediction accuracy. In particular, we look at one detailed example of how TP-AMVA, AMVA and FJ-AMVA predict single query templates. For this, we chose the Con_8 scenario and plotted the predicted response times of each method against the trace response times from Con_8, given in Figure 6. As reference we show a straight line in form of $y = x$, which depicts an optimal prediction. Predicted class response times that fall above this line are optimistic, whereas those falling below this line are of a pessimistic character.

4.2.2. Results. We give the results of our per-class prediction analysis in Figure 6. In particular, we notice that $\text{TP-AMVA}_{\text{prob}}$ predicts the majority of classes reasonably well and shows a slightly pessimistic behavior for most of the remaining query templates. We do not include $\text{TP-AMVA}_{\text{stat}}$, since it shows similar, slightly more pessimistic results than $\text{TP-AMVA}_{\text{prob}}$. Looking at our extension $\text{TP-AMVA}_{\text{prob util}}$ in scatter plot four, we note that this load-dependent modification of AMVA performs best. In contrast, the standard AMVA implementation, given by the second scatter plot in Figure 6, tends to a strong pessimistic prediction behavior, as it does not account for the variable threading level in each query template. For $\text{AMVA}_{\text{visit}}$ we observed that predictions were very optimistic, which indicates that the parameterization with the scaled service times does not improve prediction accuracy over AMVA. Interestingly, FJ-AMVA shows a diverse prediction character. On one hand, pessimistic predictions can be explained due to the summation term in the FJ-AMVA equation that produces higher response

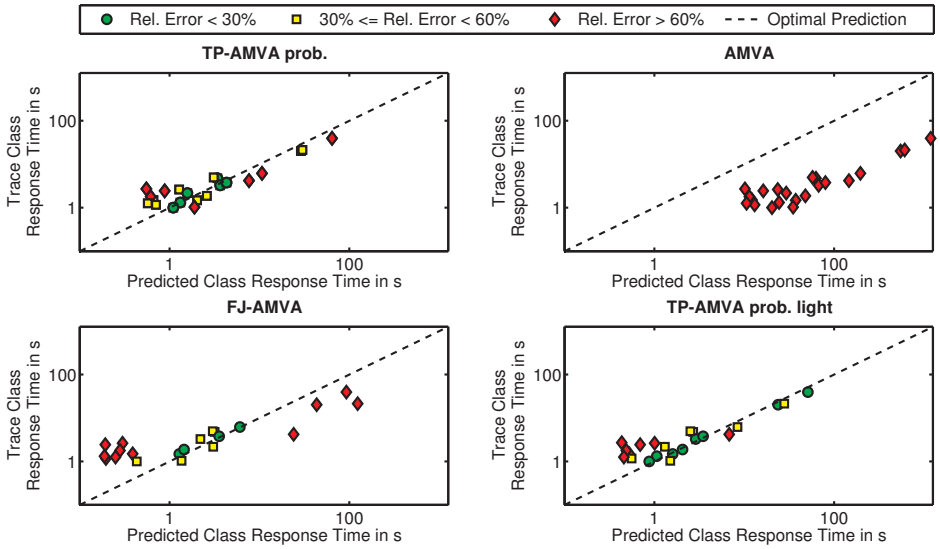


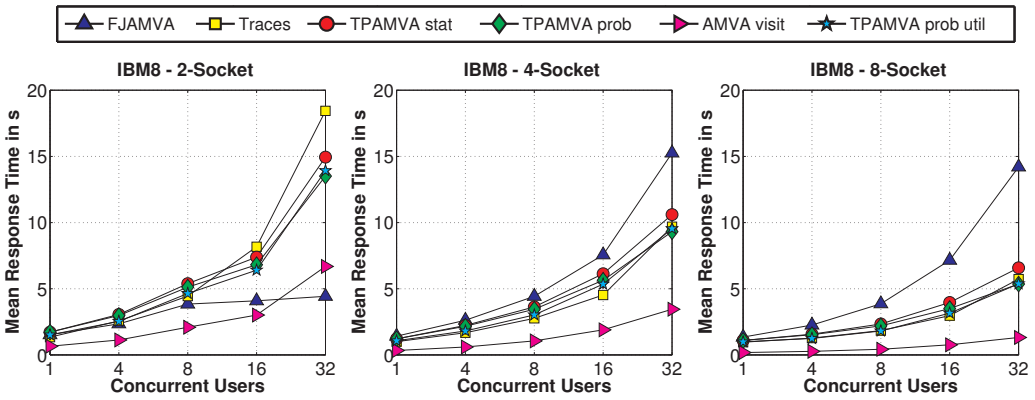
Fig. 6. Comparison of predicted per-class Response Times with per-class Trace Response Times from 8 User Scenario on the IBM4 4-socket default NUMA Configuration (normalized by Response Times of Class 1)

times for queries with high parallelism. On the other hand, optimistic predictions are caused by queries with low service times s_{ir} at each core, which we suspect is due to the non-exponentiality in s_{ir}.

We observed similar results for scenarios with 4,16 and 32 concurrent users and found that the per-class prediction accuracy across all methods is slightly decreasing the more parallel users are active. We impose this on the problem classes with high parallelism (class 1,19) and classes with long execution times (class 9,21), for which all methods produced pessimistic response times. Apart from AMVA, which always seemed to result in pessimistic predictions, we explain the optimistic predictions for short running classes due to strong contention effects, which were difficult to accurately capture by the considered methods. We found the reason for this in our traces in form of extreme blocking that caused an increase of response times for short running queries by a factor of up to 1000 under Con_{32} compared with Con_1.

4.3. Sensitivity Analysis under different Hardware Configurations

4.3.1. Evaluation Scenarios and Methodology. Having shown that TP-AMVA outperforms other methods under per-class prediction scenarios, we want to explore if our technique can be used to predict mean response times under different in-memory database system configurations. We are focusing specifically on the three in-memory database systems IBM4, IBM8 and HP8, introduced in Section 4.1.1, and conduct a sensitivity analysis to evaluate the robustness of our approximation along two different dimensions. At first, we are interested in the response time prediction accuracy when increasing the number of virtual processing cores, from 32 (2 sockets) to 64 (4 sockets) and from 64 to 128 (8 sockets). Since our IBM4 system is limited to 64 virtual cores (Hyper Threading enabled), we chose IBM8 as a reference system for this analysis. Second, we will look at the model performance across different hardware types. In that case, we keep the number of sockets fixed to four and vary the hardware type from IBM4 to IBM8 and HP8. We consider the workload scenarios from our traces with 1,4,8,16 and 32 parallel users ($\text{Con}_{1,..,32}$). Since think times in our traces are increasing with



(a) 2 Sockets (b) 4 Sockets (c) 8 Sockets

Fig. 7. Predicted Response Times across different NUMA Configurations on the IBM 8-Socket System (normalized by Response Times from 4-Socket Con_1 Scenario on IBM8)

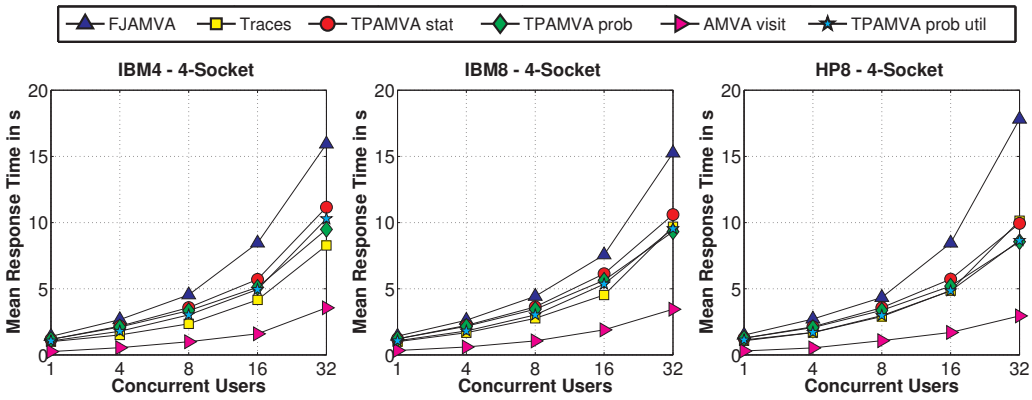
the number of parallel users, due to the sequential execution order of TPC-H query sets chosen by [Jula 2012], we use the respective trace think times for each workload scenario. In addition, we determine the mean response time W based on the per-class throughput ratios as follows:

$$W = \sum_{r=1}^{R} \frac{X_r W_r}{X}, \tag{9}$$

where the system throughput X is obtained as sum over all per-class throughputs X_r. Due to confidentiality, we normalize our results by the trace response time from Con_1 on the IBM8 4-socket configuration.

4.3.2. Results. We show our first analysis across the dimension of varying number of processing core/sockets in Figure 7. From the trace results we observe a different performance across all three system configurations, which we impose on the number of available sockets. The question is, how our analytical approximations can cope under these scenarios. Surprisingly, all three TP-AMVA variants are able to capture contention effects very accurately across all IBM8 configurations. While $\text{TP-AMVA}_{\text{stat}}$ and $\text{TP-AMVA}_{\text{prob}}$ show a slightly pessimistic character under up to 8 concurrent users, $\text{TP-AMVA}_{\text{prob util}}$ can capture contention under light load scenarios slightly better. This suggests, that our contention model in (8) improves accuracy notably. FJ-AMVA predictions tend to get more pessimistic the more parallel users are active. We found the reason for this in the response times for query classes 1,9,19 and 21, all with distinct characteristics difficult to capture. Furthermore, we observed poor results for FJ-AMVA under the 2-socket scenario, but attribute this to skewed sub-service times in the traces for this configuration. Both AMVA approximations performed poorly, since they either neglect threading levels, which was the reason to exclude the strong pessimistic results of AMVA, or use scaled service demands resulting in very optimistic response times for $\text{AMVA}_{\text{visit}}$.

We present the results of our second analysis across different hardware types in Figure 8. In general, we can observe a similar behavior for each method with respect to all three system configurations. This suggests that varying the hardware type has only little impact on the predictive capabilities. We further report the relative prediction errors across all scenarios in Table II. From the results we observe that $\text{TP-AMVA}_{\text{prob util}}$



(a) IBM4 (b) IBM8 (c) HP8

Fig. 8. Predicted Response Times across different Hardware Types with 4 Sockets (normalized by Response Times from 4-Socket Con_1 Scenario on IBM8)

Table II. Relative Error of Mean Response Time Prediction compared with Mean Trace Response Times

Method	**Virtual Processing Cores**				
	IBM8			IBM4	HP8
	32	64	128	64	64
TP-AMVA$_{\text{prob util}}$	0.13	0.09	0.04	0.19	0.05
TP-AMVA$_{\text{prob}}$	0.21	0.21	0.15	0.27	0.16
TP-AMVA$_{\text{stat}}$	0.19	0.26	0.22	0.37	0.17
FJ-AMVA	0.32	0.57	1.03	0.81	0.60
AMVA$_{\text{visit}}$	0.57	0.63	0.78	0.63	0.68
AMVA	3.55	6.39	11.00	7.48	6.16

notably improves TP-AMVA$_{\text{prob}}$, falling below a 20% error across all system configurations. While TP-AMVA$_{\text{prob}}$ and its static pendant still retain a high accuracy, FJ-AMVA predictions are too inaccurate under high load scenarios, whereas the high relative error for both AMVA variants clearly shows that both methods cannot capture contention effects properly.

4.4. Summary

From the results of our per-class evaluations and our sensitivity analysis, we conclude that AMVA, AMVA$_{\text{visit}}$ and FJ-AMVA, in its proposed form, are less 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.

5. OPTIMIZING WORKLOAD PLACEMENT

5.1. Optimization Problem and Cluster Model

Our optimization methodology aims at solving the challenge of placing analytical workloads on in-memory database clusters in a way that improves a particular objective, e.g. response times, throughputs or memory occupation, subject to given SLO and resource constraints. To represent such a cluster we consider an aggregation of database servers, each modeled by a multi-class closed QN that share a common load dispatcher, as detailed in Figure 9. Consequently, we share the workload population $\vec{N}$ amongst all servers, whereby each server maintains the same dataset locally or is connected to a shared high speed storage back-end. Recall that analytical workloads

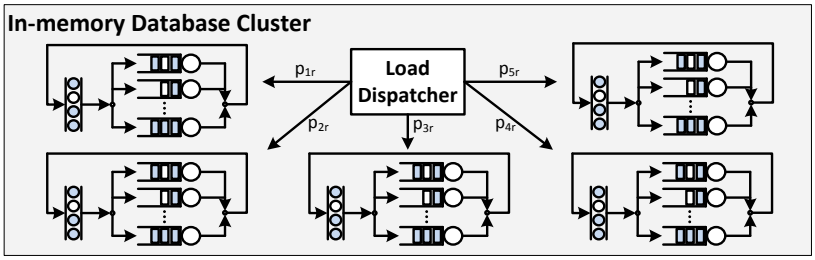


Fig. 9. Model of an in-memory Cluster Subject to Load Optimization

are read-only, and thus the dataset location has no impact on the cluster performance after datasets have been loaded into main memory.

Since we are interested into the question of how jobs should be routed from our load dispatcher to each server, we seek for optimal workload routing probabilities. Hence for our optimization model we designate p_{ir} as the probability of routing a class r request to server i and define $N_{ir} = N_r \times p_{ir},\ 1 \leq i \leq K$ as the percentage of workload that goes to server i. In the next section we will show how to model our workload routing problem with an appropriate optimization-based formulation.

5.2. Non-linear Optimization Strategy

5.2.1. Queueing Predictive Functions. We present our optimization-based formulation in (10). The objective F is generic and could include, but is not limited to, the minimization of memory consumption, response times or TCO as well as maximization of query throughputs or resource utilization. The objective is minimized by seeking routing probabilities p_{ir} that allow for near optimal workload placement:

$$opt = \min_{p_{ir}, X_{ir}, W_{ir}} F(p_{ir}, X_{ir}, Q_{ir}) \tag{10a}$$

$$\text{s.t.:} \quad U_i = \sum_r X_{ir} \frac{l_{ir}}{I_i} d_{ir},\ \forall i \tag{10b}$$

$$N_{ir} = p_{ir} N_r,\ \forall i, r \tag{10c}$$

$$Q_{ir} = X_{ir} W_{ir},\ \forall i, r \tag{10d}$$

$$\sum_r Q_{ir} = \sum_r U_{ir} \left(1 + \sum_{s=1}^{R} Q_{is}\, \delta_{irs}\, \frac{l_{is}}{I}\right),\ \forall i \tag{10e}$$

$$Q_{ir} = N_{ir} - X_{ir} Z_{ir},\ \forall r \tag{10f}$$

$$W_{ir} \geq d_{ir},\ \forall i, r \tag{10g}$$

$$\sum_i p_{ir} = 1,\ \forall r \tag{10h}$$

$$p_{ir}, X_{ir}, W_{ir} \geq 0\ \forall i, r \tag{10i}$$

$$N_{ir} > 1\ \forall i, r \tag{10j}$$

$$U_i \leq U_i^{\max},\ \forall i. \tag{10k}$$

Next to the advantage of our methodology, being able to handle a variety of objectives, one important part are the queueing predictive functions, which we integrated in form of TP-AMVA in (10b) to (10g). A problem we had to overcome was to choose the

right tradeoff between suitability for nonlinear optimization and complexity/accuracy of our three TP-AMVA expressions. Since both probabilistic versions of TP-AMVA performed best, we followed a common approach employing the less complex expression, TP-AMVA$_{\text{prob}}$, for the main optimization part and conducting a final optimization run with the more complex but also more accurate approximation, TP-AMVA$_{\text{prob util}}$. This was necessary, since TP-AMVA$_{\text{prob util}}$ causes longer optimization times due to its additional contention expressions. However, we will quantify this overhead in Section 6 and show that TP-AMVA$_{\text{prob util}}$ could still be used solely in small/medium scale optimization scenarios.

We further defined $\delta_{irs} = (N_{ir} - 1)/N_{ir} \times (l_{ir}/I_i)$ for $s = r$ and $\delta_{irs} = 1$ in case of $s \neq r$. This accounts for the Bard-Schweitzer approximation as well as the probabilistic expression of TP-AMVA, both introduced in Section 3.3 and 3.4. We further set a minimum workload of 1 job per class per server (10j), since the solution of queueing models for $N_r < 1$ is not defined. In addition, we added utilization constraints in form of $U_i^{\max}$ and ensure correct routing probabilities with (10h). From a performance point of view, our method uses less variables compared with FJ-AMVA, which would introduce at least $(I-1)\,K \times R$ additional binary variables to sort the response times for I processing cores, K servers and R classes. Since our optimization problem is non-convex, 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.

5.2.2. Minimization of Memory Occupation. We are now interested in applying our generic methodology to an important optimization problem that considers the minimization of memory consumption to prevent memory exhaustion and potential swapping inside an in-memory database cluster. We will also demonstrate the ease of integrating an additional memory occupation model into our optimization-based formulation. To represent the above optimization problem, we chose the following objective function:

$$M_{\min} = \min_{p_{ir}, X_{ir}, W_{ir}} \quad \sum_{i=1}^{K} M_i \,, \tag{11}$$

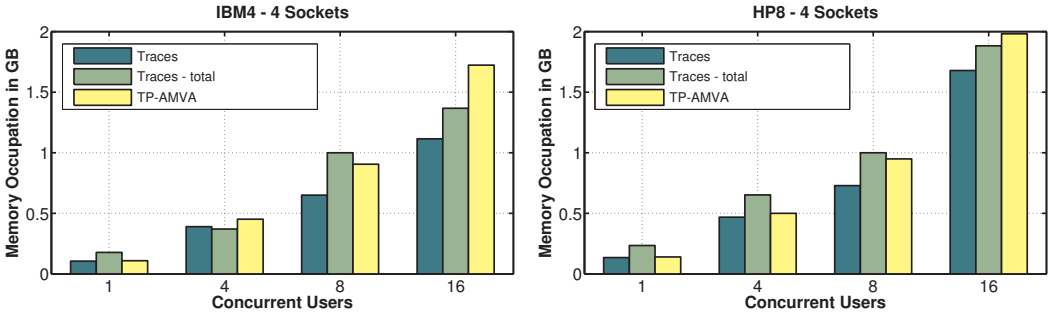
which minimizes the total sum of per-server memory occupation M_i for K in-memory database servers. Since this requires a model to estimate M_i, we developed a new memory occupation estimator of the following form:

$$M_i = \sum_r Q_{ir}\, m_{ir}, \ \forall i \tag{12}$$

and added it to the constraint set of our optimization program. In particular, for server i, we estimate M_i by multiplying the per-class mean queue length Q_{ir} of each class r with the per-class physical *peak* memory consumption m_r that is recorded in our trace logs for that class. We hereby make the conservative assumption that memory occupation grows as a function of Q_{ir} and neglect that query classes could share data residing in main memory. Additionally, we assume that forking of jobs and joining is not related to the change of memory consumption.

Finally, we added the constraint $M_i \leq M_i^{\max}, \ \forall i$ that allows us to control memory exhaustion with $M_i^{\max}$ defining the memory threshold up to which we allow servers to be exhausted.

5.2.3. Evaluating the Memory Occupation Model. Before evaluating our optimization program in the next section, we give a short analysis of our memory occupation model (12), the main part of our minimization objective in (11). For the evaluation we predicted



(a) IBM - 4 Sockets (b) HP - 8 Sockets

Fig. 10. Predicted Peak Memory Occupation under Multi-User Scenario (normalized by the 'Traces - total' value from 8 User Scenario)

the peak memory occupation with TP-AMVA$_{\text{prob light}}$ under concurrent workloads with 1 to 16 parallel users and compared it with the actual physical peak memory recorded in our traces.

We present the results of this analysis in Figure 10, which shows the peak memory occupation from the traces based on the counted per-class queue lengths Q_r multiplied with the per-class peak memory m_r as $\sum_r Q_r^{\text{counted}} \times m_r$ (Traces). We further include the total peak memory recorded from the Linux */proc/<pid>/status* file (Traces - total) and the peak memory predicted by TP-AMVA via $\sum_r Q_r^{\text{TPAMVA}} \times m_r$ (TP-AMVA). Ideally, the value for the method 'Traces' and 'Traces - total' should be the same. We can see this behavior in the similar results on the IBM and HP configuration, which suggests that our approximation in (12) is reasonably accurate. Furthermore, we attribute the gap under 8 and 16 concurrent users to outliers caused by the limited trace length of 1 hour. In addition, we explain the difference between 'Traces' and 'TP-AMVA' under Con$_{32}$ by the predicted queue length for query class 21. More specifically, we have found that class 21 causes the highest memory occupation, as shown in Figure 3(c), which thus leads to big changes in the peak memory for small increases in Q. However, we observe that the queue length predicted with TP-AMVA$_{\text{prob light}}$ gives a pretty good overall estimate of peak memory occupation in combination with (12), keeping in mind that it is generally difficult to handle outliers in an MVA framework where we do not have probabilistic measures.

6. NUMERICAL EVALUATION

6.1. Evaluation Scenarios

Throughout this section we will focus on exploring our optimization problem given in (10). Hence, we varied the number of server instances K and classes clusters R in $K, R = 4, 8, 16$. In particular, we employed k-means clustering in order to reduce our set of 22 TPC-H classes to a suitable number of clusters for the optimization process. We also refer to Appendix B for a more detailed analysis of prediction errors under class clustering. Furthermore, we defined the reference workload based on 22 classes in $N = 176K$ (light load, 8 concurrent users $\times$ 22 classes) and $N = 352K$ (heavy load, 16 concurrent users). Our class cluster populations N_r are obtained by splitting N across all class clusters depending on the amount of queries falling into a cluster. Finally, we set memory constraints to affect the workload placement: $M_i^{\max} = 512$GB for $i \leq K/2$ and $M_i^{\max} = 256$GB for $i > K/2$.

Table III. Memory Occupation and Optimality Gap†.

Inst.		Memory (GB)		Gap (%)		Max Mem.	
K	R	fm	ip	fm	ip	fm	ip
Light Load, $N = 176K$							
4	4	173.05	170.72	1.71	0.37	174.89	171.22
4	8	181.98	181.98	2.91	2.91	187.71	182.70
4	16	183.62	183.62	8.73	8.73	189.95	184.39
8	4	333.58	333.58	5.34	5.34	370.58	338.62
8	8	355.22	354.78	9.05	8.94	419.30	355.16
8	16	363.09	357.75	11.57	10.25	364.20	357.75
16	4	659.59	659.59	7.30	7.30	772.08	668.93
16	8	712.73	702.90	11.70	10.46	714.48	705.08
16	16	719.87	709.13	12.17	10.84	719.87	711.04
Heavy Load, $N = 352K$							
4	4	489.12	489.12	2.20	2.20	763.21	626.91
4	8	512.13	512.13	13.04	13.04	585.47	577.75
4	16	514.46	512.55	19.08	18.78	668.65	586.53
8	4	795.67	795.67	15.67	15.67	1196.21	808.32
8	8	920.52	912.14	26.08	25.40	1115.73	925.11
8	16	932.14	923.86	27.85	27.20	960.67	939.01
16	4	1568.28	1568.28	21.38	21.38	1575.04	1728.64
16	8	1949.40	1772.82	N/A*	24.77	N/A*	1902.88
16	16	N/A*	1805.46	N/A*	26.01	N/A*	1810.06

†Gap between best solver solution and lower bound of SCIP

*No solution found within given timeout of 1800s

6.2. Evaluation Methodology

We compare the minimization of memory swapping for two interior point based local search methods fm (Matlab's fmincon) and ip: (ipopt, [Waechter and Biegler 2006]), shipped with the OPTI Toolbox [Currie and Wilson 2012]. We particularly chose fm and ip to cope with non-linear constraints in our optimization based formulation. Since both methods rely on local solvers, we compared their multi-start based solution with the solutions found by two global solvers, scip [Achterberg 2009] and bmibnb [Loefberg 2004], as detailed in Appendix C. However, we observed that the upper bounds found by both global solvers tend to converge within only a few iterations. This indicates that multi-start based approaches are a suitable choice for solving our optimization model. Consequently, we use the fastest (scip) of the two global solvers to provide a lower bound on our optimization problem. This allows us to compute an optimality gap for fm and ip.

We implemented our approaches in MATLAB using the modeling language YALMIP [Loefberg 2004]. Our scenarios were evaluated on an Intel Core i7 CPU with 2.40GHz and 8 physical cores. To cope with different local optima, we randomized $P = 50$ initial points for every tuple $(K, R, N/K)$ and ran fm and ip using our own multi-start implementation. In addition, we report the mean execution time and its standard deviation across all P local solver runs. More specifically, we exclude the YALMIP processing overhead and only report the actual solver time spent by fm and ip. We further set a timeout of 1800 seconds to understand the performance at short time scales.

6.3. Results

6.3.1. Minimization of Memory Occupation. We present the results of our analysis in Table III. We observe that our methods fm and ip produce similar results regarding the memory occupation M for instances up to 8 servers and 4 classes. We explain this due to the same algorithm that is used to solve the queueing models. However, we found that fm lacks under scenarios with more than 8 servers and 8 classes, which we attribute to

Table IV. Optimization Times in seconds†.

Instances		**Mean**		**StdDev**		**Mean**		**StdDev**	
K	R	fm	ip	fm	ip	fm	ip	fm	ip
		Light Load				Heavy Load			
4	4	0.53	0.89	0.16	0.57	3.12	4.36	2.37	5.38
4	8	1.45	1.28	0.51	0.98	21.13	1.55	11.02	1.31
4	16	5.89	1.62	4.69	0.50	142.93	3.72	59.56	1.77
8	4	2.45	1.12	5.14	0.23	35.76	4.28	15.49	1.17
8	8	36.25	2.62	21.72	1.46	194.14	4.60	46.16	1.53
8	16	393.34	10.29	121.68	4.28	463.52	4.84	160.39	1.13
16	4	32.84	4.82	32.55	2.07	138.49	2.35	37.49	1.49
16	8	335.39	8.68	177.32	3.37	N/A*	2.80	N/A*	0.70
16	16	N/A*	58.07	N/A*	20.86	N/A*	28.48	N/A*	23.07

†Mean optimization time across all multi start runs

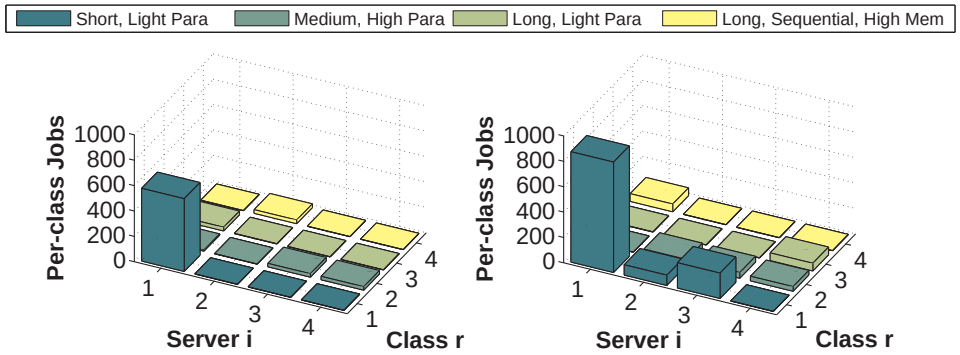
*Optimization time exceeded the given timeout of 1800s

the increased optimization time fm requires to converge to a local optimum. We were also interested in the variability across found solutions and report the worst local optimum found by fm and ip in the rightmost columns of Table III. Under both light and high load we noticed differences between the best and worst found solution of up to 16% under low load (K=8,R=8) and 36% under high load (K=4,R=4). We attribute the higher gap under heavy load scenarios to the increased workload that introduces more possibilities of being distributed amongst all servers.

We further report the optimality gap between the best found solution of our methods fm and ip compared with the lower bound found by scip in form of $|m - \text{scip}_{\text{lower}}|/m \times 100$, where $m \in \{\text{fm,ip}\}$. We noticed that under light load the possible improvements of solutions found by fm and ip fall below 13%. Under heavy load the difficulty of finding a global solution rises. We observe this through an increase of our optimality gap for ip by a factor between 2.15 (4,16) and 5.95 (4,4) compared with the respective light load scenario.

6.3.2. Optimization Times. To get an idea about the complexity of our optimization problem, we report the mean optimization times across all multi-start runs for fm and ip together with their respective standard deviations in Table IV. We observed a large gap in mean optimization times between fm and ip, which is due to the fast C++ implementation of ipopt. We also note that for method fm high load scenarios seem to be more difficult to solve, since utilization and memory constraints are more likely to be violated. Furthermore, we found that fm was unable to complete a single run within the given timeout of 1800 seconds for instances with 16 servers and 8 classes under low load as well as 8 and 16 classes under heavy load. In contrast, ip retains short optimization times, more or less independent from the actual load. This is why we explored the maximum number of servers that ip can optimize when limited to 4 customer classes and found that instances up to 512 servers could be solved in under 1000 seconds per single run.

6.3.3. Workload Placement. Another question we wanted to address is how our optimization program handles workload placement. We therefore investigated the instance with 4 servers and 4 classes under light and heavy load. Figure 11 shows the workload distribution we obtained with method ip after optimization as well as the query characteristics regarding service demand and parallelism. Under light load server 2 uses 125GB, whereas the other servers show a memory occupation of ≈15GB, meaning no constraints are violated. However, the heavy load situation looks different. The memory bound portion of our workload (class 4) is now dispatched to servers with a memory constraint of 512GB, in this case server 1 (using 340GB), since server 3 and



(a) Light Load (b) Heavy Load

Fig. 11. Optimized Placement of Workload under light and heavy Load

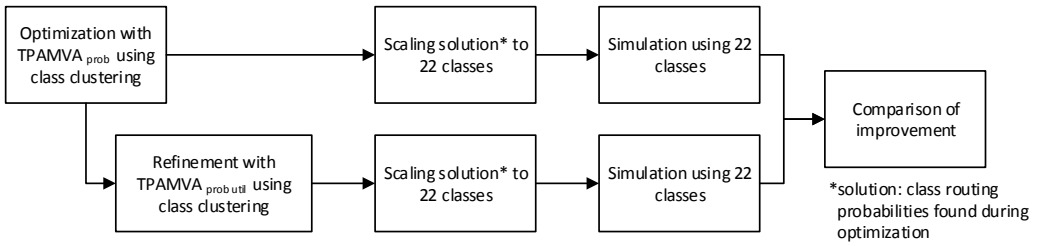


Fig. 12. Methodology for Optimization Refinement and Evaluation against Simulation

4 are limited to 256GB. We also noticed that under light load, in Figure 11(a), classes with higher memory occupation, such as classes 2 and 4, are placed in a way that minimizes interference with other classes: class 4 on server 2, class 2 on server 3 and 4. Note that at least one job per class is placed on each server, since closed queueing networks are not defined for $N_r < 1$. Under heavy load resources on server 2 to 4 are fully utilized. The effect we can observe now is that the class with the highest memory occupation (class 4) is isolated on server 1 and collocated with a class of lowest impact (class 1) due to the remaining workload that cannot be handled by server 2 to 4. From this we conclude that our optimization program handles resource constraints appropriately.

6.3.4. Optimization Refinement and Validation. At last we look into further refining our optimization results as mentioned in Section 5.2.1. To better understand this refinement step we detail our methodology in Figure 12. In particular, we took the best solution found by method ip based on TP-AMVA$_{\text{prob}}$ as a starting point for a final run with TP-AMVA$_{\text{prob util}}$. We then reversed the class clustering applied during the optimization process and used simulation to quantify the actual improvement that can be achieved by a refinement run with TP-AMVA$_{\text{prob util}}$. Consequently, we determined the optimal workload distribution found by both TP-AMVA models, and used each as input for a final simulation run. We then computed the percentage of reduction in simulated memory occupation of TP-AMVA$_{\text{prob util}}$ over TP-AMVA$_{\text{prob}}$. We detail these results in Figure 13 for the more relevant heavy load scenario. In fact, the refinement step reduces the simulated memory occupation by approximately 7% across all scenarios.

(a) 4 Servers (b) 8 Servers (c) 16 Servers

Fig. 13. Improvement in simulated Memory Occupation using optimal Workload Placement found by TP-AMVA$_{\text{prob util}}$ compared with TP-AMVA$_{\text{prob}}$ as baseline

This clearly works in favor for our approach. Admittedly, we noticed that during our experiments using TP-AMVA$_{\text{prob util}}$ could slow down the solution process compared with TP-AMVA$_{\text{prob}}$ by a factor up to 20 due to its additional nonlinear expressions. Nonetheless, we found that we were still able to use TP-AMVA$_{\text{prob util}}$ during the entire optimization process for scenarios up to 8 servers and 8 job classes. For larger scenarios with up to 512 servers however, we recommend to use TP-AMVA$_{\text{prob}}$, and if possible conduct a final run with TP-AMVA$_{\text{prob util}}$.

6.3.5. Summary. Summarizing the results, we have shown based on empirical evidence:

— that on our optimization-based formulation multi-start based local search strategies achieve a good optimality compared with global solvers
— that class aggregation can help to improve optimization times while retaining a reasonable level of accuracy, in particular in combination with TP-AMVA$_{\text{prob util}}$
— that our optimization methodology appropriately handles resource constraints under workload placement scenarios on in-memory database systems
— and finally that fast interior-point based methods, such as ipopt, can be used for optimization scenarios up to 512 servers and 4 classes, before optimization times exceed the set timeouts.

7. RELATED WORK

While more than a decade ago research introduced fundamental cost models for the entire memory hierarchy in a database system [Manegold et al. 2002], nowadays on-demand provisioning of these systems is driving research further into database optimization and encourages the use of queueing networks [Osman and Knottenbelt 2012].

Multi-tenancy and Scheduling

In [Elmore et al. 2013] 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, this work does not consider variable threading levels and puts focus mainly on transactional workloads. Workload characterization and response time prediction via non-linear regression techniques for in-memory databases are proposed in [Schaffner et al. 2011a]. The authors derive

tenant placement decisions by employing first fit decreasing scheduling, but evaluate on small scale only. [Lang et al. 2012] 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 service level objectives (SLOs), but relies on brute force solvers and ignores OLAP workloads. [Kelly et al. 2008] present three simple operational laws based on open queues. Their analysis method applies to scaling decisions for multi-core network servers and is validated on real HP systems. This method depends on live-monitoring and neglects job class information.

Optimization

[Zhang et al. 2014] consider hardware and workload heterogeneity in Google's cloud data centers to optimize energy consumption by dynamically adjusting allocated resources. The authors use clustering approaches to reduce large heterogeneous workloads with distinct resource demands in CPU and memory and combine probabilistic expressions of an open queueing model with a mixed-integer optimization approach to solve their provisioning problem. However their methodology requires heuristics for finding a good solution. In [Das et al. 2013] 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. [Almeida et al. 2006; Rogers et al. 2010] introduce frameworks for non-linear cost optimization regarding SLA violations and resource usage, applied to web service based applications and cloud databases. However, regarding per-class CPU resource cost both approaches only focus on service demands and CPU cycles, but neglect variable threading of workload classes. In addition, [Rogers et al. 2010] considered only the first 5 query templates of the TPC-H benchmark at small scale factors, whereas our workload characterization illustrates the importance of the remaining queries and considers a scale factor of 100. [Li et al. 2010] 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 parallelism.

Prediction Models

[Duggan et al. 2011; Wu et al. 2013; Suri et al. 2007] 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 patterns and are evaluated for small numbers of jobs and batch workloads only. Ignored by the latter, thread-level parallelism is addressed by [Kraft et al. 2012] and [Alomari and Menascé 2014], but despite using similar techniques, their approaches are computationally expensive or rely on exponential service time distributions. An interesting aspect was considered in [Dan et al. 1988]. The authors used probabilities to model data and resource access conflicts in database systems to describe contention effects more accurately, however did not account for the extensive threading levels that occur in analytical workloads.

8. CONCLUSIONS AND FUTURE WORK

In this paper we have made several contributions that include a novel analytic response time approximation for in-memory databases, which models thread-level fork join and per-class memory occupation. 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. In addition, we have developed an extensible optimization methodology that can be used to optimize workload placement for clusters of in-memory database systems in cloud infrastructures.

Next to implementing our provisioning framework in a real in-memory database management system, possible directions for future work include the modeling of resource contention under multi-tenancy, where client workloads are of transactional and operational characters or are based on differently sized datasets. Other directions focus on resource allocation challenges, such as optimizing CPU and memory resources for multiple co-located tenant databases on multi-socket systems in order to provide performance guarantees.

REFERENCES

ACHTERBERG, T. 2009. SCIP: Solving constraint integer programs. *Mathematical Programming Computation 1,* 1, 1–41.

ALMEIDA, J., ALMEIDA, V., ARDAGNA, D., FRANCALANCI, C., AND TRUBIAN, M. 2006. Resource Management in the Autonomic Service-Oriented Architecture. In *Proceedings of the 3rd International Conference on Autonomic Computing (ICAC '06)*.

ALOMARI, F. AND MENASCÉ, D. 2014. Efficient Response Time Approximations for Multiclass Fork and Join Queues in Open and Closed Queuing Networks. *IEEE Transactions on Parallel and Distributed Systems 25,* 6, 1437–1446.

BASKETT, F., CHANDY, K. M., MUNTZ, R. R., AND PALACIOS, F. G. 1975. Open, Closed, and Mixed Networks of Queues with Different Classes of Customers. *Journal of the ACM 22,* 2, 248–260.

CHENG, W. C. AND MUNTZ, R. R. 1996. Bounding Errors Introduced by Clustering of Customers in Closed Product-form Queuing Networks. *Journal of the ACM 43,* 4, 641–669.

CURRIE, J. AND WILSON, D. I. 2012. OPTI: Lowering the Barrier Between Open Source Optimizers and the Industrial MATLAB User. *Foundations of Computer-Aided Process Operations*.

DAN, A., TOWSLEY, D., AND KOHLER, W. 1988. Modeling the effects of data and resource contention on the performance of optimistic concurrency control protocols. In *Proceedings of the Fourth International Conference on Data Engineering (ICDE '88)*.

DAS, S., NARASAYYA, V. R., LI, F., AND SYAMALA, M. 2013. CPU Sharing Techniques for Performance Isolation in Multitenant Relational Database-as-a-Service. *Proceedings of the VLDB Endowment 7,* 1, 37–48.

DUGGAN, J., CETINTEMEL, U., PAPAEMMANOUIL, O., AND UPFAL, E. 2011. Performance prediction for concurrent database workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*.

DUGGAN, J., PAPAEMMANOUIL, O., ÇETINTEMEL, U., AND UPFAL, E. 2014. Contender: A Resource Modeling Approach for Concurrent Query Performance Prediction. In *Proceeding of the 17th International Conference on Extending Database Technology (EDBT '14)*.

ELMORE, A. J., DAS, S., PUCHER, A., AGRAWAL, D., EL ABBADI, A., AND YAN, X. 2013. Characterizing tenant behavior for placement and crisis mitigation in multitenant DBMSs. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*.

FÄRBER, F., CHA, S. K., PRIMSCH, J., BORNHÖVD, C., SIGG, S., AND LEHNER, W. 2011. SAP HANA database: data management for modern business applications. *SIGMOD Record 40,* 4, 45–51.

JULA, A. 2012. Multicore scalability and NUMA effects on HDB with SAP-H data and TPC- H queries. Tech. rep., SAP.

KELLY, T., SHEN, K. S. K., ZHANG, A., AND STEWART, C. 2008. Operational Analysis of Parallel Servers. In *Proceedings of the 16th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '08)*.

KRAFT, S., CASALE, G., JULA, A., KILPATRICK, P., AND GREER, D. 2012. WIQ: Work-Intensive Query Scheduling for In-Memory Database Systems. In *Proceedings of the fifth International Conference on Cloud Computing (CLOUD '12)*.

LANG, W., SHANKAR, S., PATEL, J. M., AND KALHAN, A. 2012. Towards Multi-tenant Performance SLOs. In *Proceedings of the 28th International Conference on Data Engineering (ICDE '12)*.

LAZOWSKA, E., ZAHORJAN, J., GRAHAM, G., AND SEVCIK, K. 1984. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice Hall.

LI, H., CASALE, G., AND ELLAHI, T. 2010. SLA-driven planning and optimization of enterprise applications. In *Proceedings of the first joint WOSP/SIPEW international conference on Performance engineering (WOSP/SIPEW '10)*.

LOEFBERG, J. 2004. Yalmip : A toolbox for modeling and optimization in MATLAB. In *Proceedings of the International Symposium on Computer-Aided Control Systems Design (CACSD '04)*.

MANEGOLD, S., BONCZ, P., AND KERSTEN, M. L. 2002. Generic database cost models for hierarchical memory systems. In *Proceedings of 28th International Conference on Very Large Data Bases (VLDB '02)*.

MASTROIANNI, C., MEO, M., AND PAPUZZO, G. 2013. Probabilistic Consolidation of Virtual Machines in Self-Organizing Cloud Data Centers. *IEEE Transactions on Cloud Computing 1,* 2, 215–228.

MOLKA, K., CASALE, G., MOLKA, T., AND MOORE, L. 2014. Memory-aware sizing for in-memory databases. In *Proceedings of the IEEE/IFIP Network Operations and Management Symposium (NOMS '14)*.

OSMAN, R. AND KNOTTENBELT, W. J. 2012. Database System Performance Evaluation Models: A Survey. *Performance Evaluation 69,* 10, 471–493.

ROGERS, J., PAPAEMMANOUIL, O., AND CETINTEMEL, U. 2010. A generic auto-provisioning framework for cloud databases. *Workshops Proceedings of the 26th International Conference on Data Engineering (ICDEW '10)*.

ROLIA, J., CASALE, G., KRISHNAMURTHY, D., DAWSON, S., AND KRAFT, S. 2009. Predictive Modelling of SAP ERP Applications: Challenges and Solutions. In *Proceedings of the 1st International Workshop on Run-time mOdels for Self-managing Systems and Applications (ROSSA '09)*. VALUETOOLS'09.

SAP. 2013. SAP HANA Performance: Efficient Speed and Scale-Out for Real-Time Business Intelligence.

SCHAFFNER, J., ECKART, B., JACOBS, D., SCHWARZ, C., PLATTNER, H., AND ZEIER, A. 2011a. Predicting In-Memory Database Performance for Automating Cluster Management Tasks. In *Proceedings of the 27th International Conference on Data Engineering (ICDE '11)*.

SCHAFFNER, J., ECKART, B., SCHWARZ, C., BRUNNERT, J., JACOBS, D., AND ZEIER, A. 2011b. Towards Analytics-as-a-Service Using an In-Memory Column Database. In *New Frontiers in Information and Software as Services*. Vol. 74. 257–282.

SCHWEITZER, P. J. 1979. Approximate analysis of multiclass closed networks of queues. In *Proceedings of International Conference on Stochastic Control and Optimization*. Amsterdam, 25–29.

SURI, R., SAHU, S., AND VERNON, M. 2007. Approximate Mean Value Analysis for Closed Queuing Networks with Multiple-Server Stations. In *Proceedings of the Industrial Engineering Research Conference*.

VARKI, E. 2001. Response time analysis of parallel computer and storage systems. *IEEE Transactions on Parallel and Distributed Systems 12,* 11, 1146–1161.

WAECHTER, A. AND BIEGLER, L. T. 2006. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming 106,* 1, 25–57.

WU, W., CHI, Y., HACIGÜMÜS, H., AND NAUGHTON, J. F. 2013. Towards Predicting Query Execution Time for Concurrent and Dynamic Database Workloads. *Proceedings of the VLDB Endowment 6,* 10, 925–936.

ZHANG, Q., ZHANI, M. F., BOUTABA, R., AND HELLERSTEIN, J. L. 2014. Dynamic Heterogeneity-Aware Resource Provisioning in the Cloud. *IEEE Transactions on Cloud Computing 2,* 1, 14–28.

APPENDIX

A. ESTIMATION OF SERVICE DEMANDS FOR FJ-AMVA

This section describes how we have estimated FJ-AMVA parameters. In addition to the core activity, introduced in Section 4.1.2, our traces record the number of threads T_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 [Kraft et al. 2012], and thus prompted us to extract it from the raw traces. We illustrate this in Figure 14 Case 2a, which lists all 7 threads that belong to our exemplary job, introduced in Section 4.1.2. 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 used s_r^t for its parameterization in our experiments. Additionally, FJ-AMVA assumes that the number of per-class tasks T_r is not bigger than the number of available processing cores I. However, for some classes and also for our example, with $T = 7$ 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 $T_r > I$, $s_r^t \leq 0.2s$ for $t > I$. Since

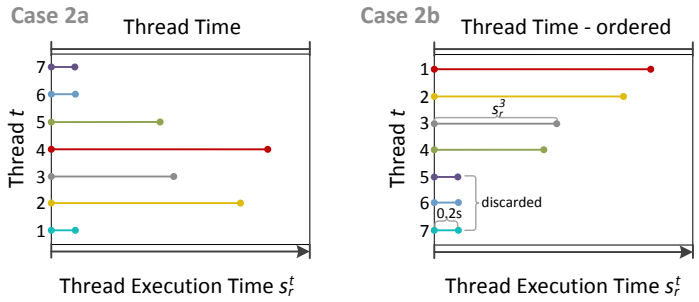


Fig. 14. Service Demand Estimation for an OLAP Query

[Jula 2012] used a sampling interval of 0.2 seconds to collect the traces, these threads can be ignored because their execution time falls under the sampling inaccuracy.

B. EFFECTS OF CLASS CLUSTERING

As part of the evaluation of our optimization methodology in Section 6 we give here an additional analysis of the class clustering model that we used. In particular, we are interested in how the performance measures of our queueing model, such as system utilization U, memory occupation M, mean response time W and system throughput X are affected when parameterizing TP-AMVA with aggregated class parameters. We therefore clustered our set of $R = 22$ TPC-H classes with k-means (a priori normalized by z-score) across the two dimensions parallelism l_r, service demand d_r, depicted for the cluster sizes $C = 2, 4, 8$ in Figure 15. This required us to redefine the workload $(\vec{N}, \vec{Z})$, based on the original 22 class scenario with the number of per-class jobs defined by $N_r = \text{Con}_i$ and the total number of jobs defined by $N = \sum_r^R N_r$. Subsequently, we estimated the number of jobs per class cluster N_c as discussed in [Cheng and Muntz 1996] according to the frequency of each class occurring in a cluster, which in our case is $N_c = \sum_{r,r\in c} N_r$. In addition, we estimated the per-cluster think times Z_c under consideration of the response time law [Lazowska et al. 1984], using the trace throughputs and response times from Con_i:

$$Z_c = \frac{N_c}{\sum_{r,r\in c} X_r} + \frac{1}{c_{\text{size}}} \sum_{r,r\in c}^{R} W_r, \quad \forall c, \tag{13}$$

where c_{size} denotes the number of classes falling into class cluster c.

We determined the relative error of TP-AMVA$_{\text{prob}}$ under class clustering compared with a reference run using 22 classes under workload scenarios with 1,4,8,16 and 32 parallel users. Since we observed similar prediction errors under all scenarios, we only give the results of our class clustering analysis for 4 and 16 parallel users in Table V. As expected, the more classes we use the more accurate the prediction gets. However, we note that reducing the original class set from 22 classes down to 8 class clusters only slightly effects the prediction accuracy, whereas further clustering increases prediction errors notably. While errors using 4 class classes are still acceptable, we do not recommend to use less clusters on our dataset, since this results in utilization and memory occupation estimates at an approximate error of 50%. Based on these results, we decided to consider 4,8 and 16 classes for the evaluation of our optimization program in Section 6.

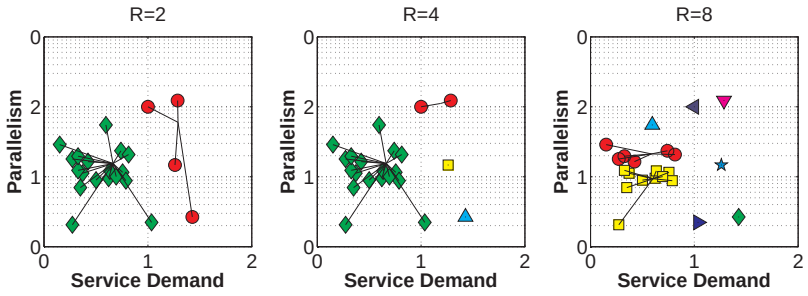


Fig. 15. Normalized Query Classes for different Numbers of k-means Clusters (logarithmic scaling)

Table V. Relative Prediction Error† under Class Clustering compared with 22 Class Scenario on Single Server

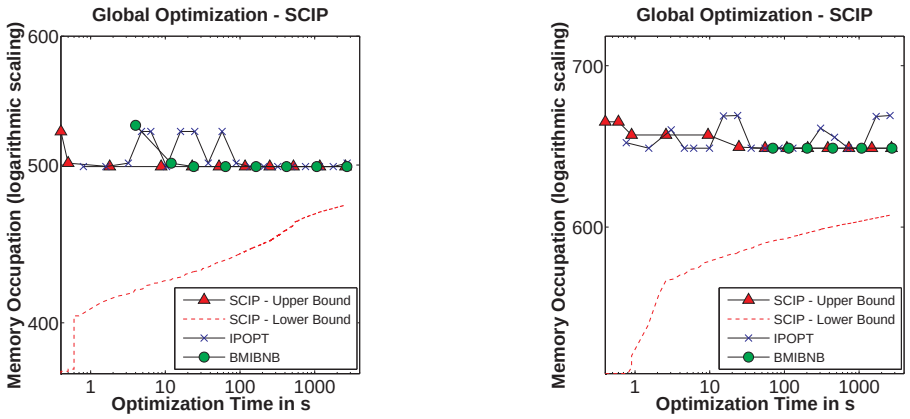
Clusters C	U		M		W		X	
	Con4	Con16	Con4	Con16	Con4	Con16	Con4	Con16
2	0.46	0.59	0.46	0.54	0.09	0.23	0.01	0.02
4	0.04	0.02	0.05	0.18	0.07	0.22	0.01	0.01
8	0.00	0.00	0.01	0.01	0.01	0.01	0.00	0.00
16	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
20	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
22	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

†Relative prediction error: |estimate-reference|/reference

Notation: Utilization U, Memory Occupation M, Response Time W and Throughput X

C. JUSTIFICATION FOR A MULTI START BASED OPTIMIZATION APPROACH

In Section 6.2 we gave a brief explanation on why we used local solvers in combination with a multi-start initialization approach to solve our optimization model. Here we provide the results that support this solver choice. One obvious reason for this choice is that global optimization quickly becomes intractable with a growing complexity of a non-linear optimization model. Hence, we used the local solver ip (based on ipopt) and explored how large the gap between solutions of our multi-start based local solvers compared with global solvers are. For this analysis we chose two different scenarios, given in Figure 16, which we ran until an optimality gap of 6% was reached. For both scenarios we observed that the upper bound was minimized very quickly. This means we generally do not need many iterations to achieve a good solution, which in worst case could only further improve by 6%. We impose the difficulty of further reducing the optimality gap on the large search space spanned by our decision variables. However, the results suggest that our optimization problem is of such a form that reducing the optimality gap further would have only little impact on the actual improvements and thus is a strong indicator for preferring a multi-start approach based on ipopt. We also found that bmibnb takes longer to converge than SCIP, due to its additional processing overhead. Hence for our evaluation in Section 6 we used SCIP to provide a lower bound and ipopt to determine an upper bound on our optimization problem.



(a) 6 Servers, 4 Classes (b) 32 Servers, 2 Classes

Fig. 16. Global Optimization (stopped after 6% Duality Gap reached)