An Elastic Multi-Core Allocation Mechanism for Database Systems

Simone Dominico*, Eduardo C. de Almeida*, Jorge A. Meira† and Marco A. Z. Alves*
*UFPR, Brazil – sdominico@inf.ufpr.br, eduardo@inf.ufpr.br, mazalves@inf.ufpr.br
†University of Luxembourg – jorge.meira@uni.lu

Abstract—**During the parallel execution of queries in Non-Uniform Memory Access (NUMA) systems, the Operating System (OS) maps the threads (or processes) from modern database systems to the available cores among the NUMA nodes using the standard *node-local* policy. However, such non-smart mapping may result in inefficient memory activity, because shared data may be accessed by scattered threads requiring large data movements or non-shared data may be allocated to threads sharing the same cache memory, increasing its conflicts. In this paper we present a data-distribution aware and elastic multi-core allocation mechanism to improve the OS mapping of database threads in NUMA systems. Our hypothesis is that we mitigate the data movement if we only hand out to the OS the local optimum number of cores in specific nodes. We propose a mechanism based on a rule-condition-action pipeline that uses hardware counters to promptly find out the local optimum number of cores. Our mechanism uses a priority queue to track the history of the memory address space used by database threads in order to decide about the allocation/release of cores and its distribution among the NUMA nodes to decrease remote memory access. We implemented and tested a prototype of our mechanism when executing two popular Volcano-style databases improving their NUMA-affinity. For MonetDB, we show maximum speedup of 1.53×, due to consistent reduction in the local/remote per-query data traffic ratio of up to 3.87× running 256 concurrent clients in the 1 GB TPC-H database also showing system energy savings of 26.05%. For the NUMA-aware SQL Server, we observed speedup of up to 1.27× and reduction on the data traffic ratio of 3.70×.**

I. Introduction

In modern systems with Non-Uniform Memory Access (NUMA) architecture, formed by multiple multi-core nodes, the memory hierarchy becomes more complex, composed of multiple cache levels and different memory sharing schemes among the cores. In this context, to find data affinity for multi-threaded applications is an open and highly relevant problem. Therefore, the correct mapping of threads and data over the NUMA nodes reduce data movement through the memory hierarchy also reducing the number of cache conflicts and invalidations between the threads leading to improvements on the final performance.

In this context, when Database Management Systems (DBMS) based on the Volcano query parallelism model [1] execute Online Analytical Processing (OLAP) workloads, parallelism is hidden from operators. Typically, parallelism is set in the query plan at planning time and the work is assigned to threads statically, which usually uses inaccurate memory estimation [2], [3]. In this model, the Operating System (OS) is in charge of mapping the threads or processes from database queries to as many processors and cores as possible [4]. It means that the OS traditionally tries to allocate one thread per core in a scattered way, among all the available NUMA nodes. However, if multiple threads sharing the same chunk of data are mapped far apart (e.g., onto different NUMA nodes), they may cause vast amount of data movement increasing inter-node traffic and the number of cache invalidations. On the other hand, if threads acting over private chunks of data are mapped nearby (e.g., inside the same NUMA node), they may cause a high number of cache conflicts.

In this paper, we present a processing core allocation mechanism to support the thread scheduling and data allocation across NUMA sockets. Our hypothesis is that we mitigate data movement in NUMA nodes if the OS only maps threads to an efficient sub-set of processing cores for each specific workload based on database performance states. Thus, our core allocation mechanism systematically analyses the hardware resources usage by the running threads to set up the current database performance state. Then, the mechanism decides if cores need to be allocated or released with respect to the performance state. In this sense, our proposal is orthogonal to previous state-of-the-art works, such as SAP Hana [5], that propose adaptive data and inter-socket thread placement strategies based on hardware counters (e.g., the memory intensity of threads). Nonetheless, results show that our mechanism can further improve the performance for NUMA-aware DBMS, such as SQL Server [6].

Our mechanism is implemented on top of an abstract model of the database performance states using PetriNet theory. We model performance goals (or predicates) that need to be satisfied to trigger the allocation of CPU cores, namely PetriNet Predicate/Transition (PrT), in order to bring the database to a stable performance state. Figure 1 overviews our PetriNet mechanism in the query processing ecosystem on NUMA. The PetriNet monitors the resource usage of the worker threads on top of OS kernel facilities to decide for the allocation of CPU cores (e.g., cgroups, mpstat, numactl, likwid). The correct location for the next allocation depends upon a priority queue maintained by different policies, called allocation modes (allocated cores are depicted in black).

As far as we know, we are the first to propose a mechanism that provides to the OS an efficient sub-set of cores to perform thread mapping considering the NUMA data placement statistics specific for DBMS. Overall, our main contribution in this paper are the following:

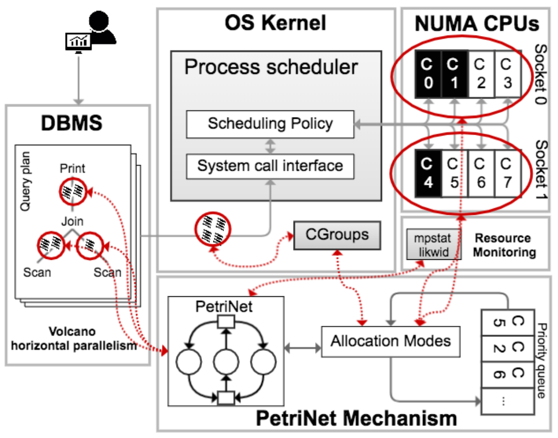


Fig. 1: OS scheduler performing the mapping of database threads supported by our PetriNet mechanism.

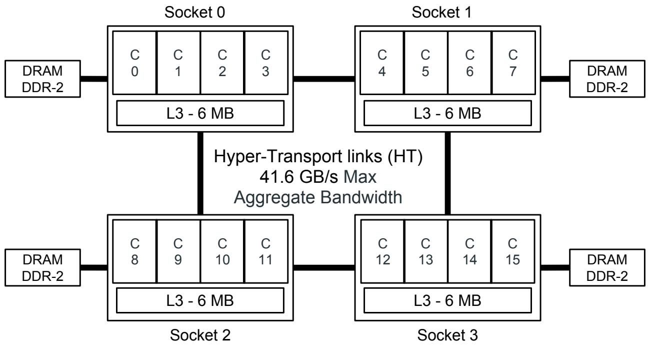


Fig. 2: Four NUMA nodes formed by Quad-Core AMD Opteron 8000 Series, interconnected by Hyper-Transport (HT) 3.x link.

Analysis of data movement in NUMA: We discuss why handing out all the cores available in the system to the OS may muck up OLAP performance. We show microbenchmark results of interconnection bandwidth usage between NUMA nodes to present our motivation.

The abstract model: We propose an abstract model for resource allocation in NUMA machines based on database performance states. The model can be easily adapted to allocate either multi-cores or remote memory in any OS and DBMS of the user choice.

The local optimum number of cores: We present the concept of "local optimum number of cores" to tackle the current OLAP workload, controlling thus the allocation and release of processing cores along the execution of queries.

An adaptive multi-core allocation algorithm: We present different allocation modes to maintain the local optimum number of cores and an adaptive algorithm decides the location for the next allocation/release taking into account the accessed memory addresses kept into a priority queue data structure.

Empirical results: We show improvements on NUMA-affinity when executing MonetDB and SQL Server with our mechanism. For MonetDB, results show consistent reduction in the local/remote per-query data traffic ratio of up to 3.87× (2.47× on average) running the TPC-H leading to speedups as higher as 1.53× (1.29× on average). With less data traffic, we observed important system energy savings of 26.05% on the overall system. For SQL Server, we observed reductions of up to 3.70× (2.57× on average) in per-query data traffic ratio and speedup gains up to 1.27× (1.14× on average).

This paper is organized as follows: next section describes the problem of moving data around through the interconnection when processing OLAP. Section III discuss the abstract model and the rule-condition-action pipeline. Section IV presents the core allocation modes and the implementation details. Section V shows empirical results and Section VI discusses related work. Section VII brings conclusions.

II. Data movement impact in NUMA machines

In this section, we discuss the data movement issues during the execution of DBMS threads in a NUMA machine. We also study a microbenchmark to deeply understand what happens when moving data from the NUMA nodes to cores/threads working on data. Our microbenchmark consists of the TPC-H query 06 (Q6), because of its data access pattern with high spatial locality [7]. We study two versions of Q6, the original SQL and the hand-coded C language version.

A. NUMA Architecture and Thread Scheduling

Nowadays, NUMA is widely available in multi-socket machines. For such systems, mapping threads into processors near where the data resides helps improving overall system performance [8]. Figure 2 presents a diagram of a NUMA architecture using AMD Opteron 8000 processors with a set of cores and a DDR-2 memory bank attached to each node. This specific NUMA architecture will be used in our evaluations, and thus it is important to understand its details. Each NUMA node is attached to a memory bank, the nodes and the memories are interconnected using Hyper-Transport (HT) 3.x links. The memory access latency varies accordingly to the distance between the node (where the core is located) and the memory being accessed. Therefore, access to remote memories imposes higher latency than local memories due to the interconnection.

In Linux, the OS scheduler may determine from which NUMA node the kernel will allocate memory with respect to load balancing scheduling policies[1]. Ideally, we want the database threads to access local memory, which occurs in the default policy, called *node-local*. We assume in this paper the Volcano-style horizontal parallelism implemented by most of the DBMS where the execution of an operator at a time spans many threads [2], [1]. We also assume that a non-NUMA DBMS hands over to the OS scheduler the control over data and thread locality, unlike a NUMA-aware DBMS that explicitly assigns worker threads to cores (discussed in Section VI).

[1]https://www.kernel.org/doc/Documentation/vm/numa _memory_policy.txt

SQL:

```
SELECT
   sum(l_extendedprice * l_discount) as revenue
FROM
   LINEITEM
WHERE
   l_shipdate >= date '1997-01-01'
AND l_shipdate < date '1997-01-01' + interval '1' year
AND l_discount between 0.07 - 0.01 and 0.07 + 0.01
AND l_quantity < 24;
```

MAL:

```
X_1:= algebra.thetasubselect(l_quantity)
X_2:= algebra.subselect(l_shipdate,X_1)
X_3:= algebra.subselect(l_discount,X_2,)
X_4:= algebra.projection(X_3,l_extendedprice)
X_5:= algebra,projection(X_3,l_discount)
X_6:= [*](X_4,X_5)
X_7:= aggr.sum(X_6)
```

C:

```
static double tpch_query6(int lineitemSize, double *__restrict__ p_quanEty, double *__restrict__
p_extendedprice, double *__restrict__ p_discount, int *__restrict__ p_shipdate, double
sum_revenue, double discount) {
     for (int i = 0; i < lineitemSize; i++) {
          if (p_shipdate[i] >= 19970101 && p_shipdate[i] < 19980101) {
               if (p_discount[i] >= (discount - 0.01) && p_discount[i] <= (discount + 0.01)) {
                    if (p_quanEty[i] < 24) {
                         sum_revenue += p_extendedprice[i] * p_discount[i]; }}}}
     return sum_revenue; }
```

Fig. 3: SQL and C code versions of the TPC-H Q6 and the query plan written in Monet Assembly Language (MAL).

On the thread startup, at the first use of a memory page, called first touch, the OS determines its NUMA node location. These memory pages (i.e., data from the threads) can be moved around during the execution, as well as the thread can be moved around to a different NUMA node for performance purposes. During the thread creation process, the OS scheduler attempts to leave them on remote nodes balancing thus the CPU load. This way, these CPUs far-apart will not share cache memories. This is the case of a typical OLAP workload with many threads spread all over the nodes in which data movement occurs to catch up with threads in remote nodes. The effect of data movement is more compelling when large structures are accessed frequently with lots of data shared among the threads. In the rest of this section, we discuss this effect that motivates our work.

B. TPC-H Q6 evaluation

In this subsection we study the data movement effect on NUMA showing performance and resource usage statistics.

1) **Impact of remote memory access:** Our first experiment aims to show that the current OS scheduler is not optimal for NUMA systems executing DBMS queries, leading to a high number of remote memory accesses even when querying over the same data concurrently. For this experiment ran the Q6 microbenchmark on the 4-node quad-core AMD Opteron machine depicted on Figure 2. Figure 3 shows the SQL and C implementations of Q6 and the query plan. We compare the data/thread scheduling performed by the unmodified OS policy to a controlled execution with predefined sparse or dense thread/data affinity. We also compare the hand-coded C language version of Q6 to the SQL version on MonetDB of the same query. Both versions were created using 1 GB scale factor of raw data upon different numbers of concurrent

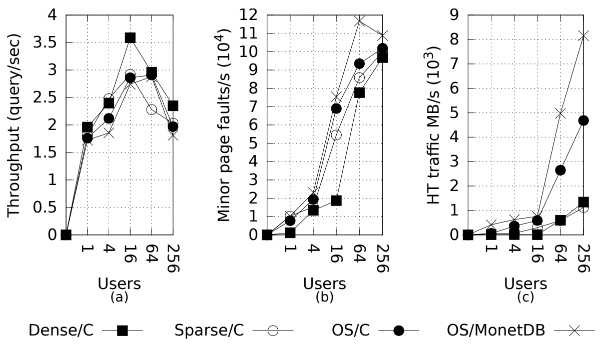


Fig. 4: TPC-H Q6 execution with an increasing number of concurrent clients.

clients. For details about the experimental setup see Section V.

We experimented the C language hand-coded version of the original Q6 written using Posix threads (*pthreads*) for two reasons: (1) to state the baseline of raw performance (to establish a near-to-limit performance) and; (2) to analyze if setting the affinity of *pthreads* makes the OS improving data locality. We only implemented the columns that are part of the query and the operations were made parallel to mimic the Volcano model of multiple threads. The *pthreads* were mapped to the cores using two different policies (dense and sparse) setting the affinity of a thread to one or all cores (*pthread_setaffinity_np()*). In the dense mode all the *pthreads* are sent to the same node, whereas in the sparse mode *pthreads* are spread to cores on different nodes. These affinity modes follow the idea of incremental allocation presented in [9], [10].

Figure 4 compares the OS scheduling for MonetDB worker threads (OS/MonetDB) and the hand-coded C language version of Q6 (OS/C). It also presents the scheduling for the C-code using two preset data affinities (Sparse/C and Dense/C). As expected, results show the same effect present on the C-code and the SQL-code versions: the traffic of interconnection increases with the number of concurrent clients, because the OS imposes a load balance during the execution of the code. However, the OS does a better job finding data affinity with the C-code rather than DBMS. This happens, because the C-code creates multiple threads from a single program, while the SQL version generates multiple threads for every operator in the query plan using the Volcano horizontal parallelism creating a more complex iteration system. This reflects in the query throughput and the increasing number of page faults when increasing concurrency (Figures 4(a) and (b)). The difference of interconnection usage between both versions (SQL and C) varies from 100× with a single client and 8× when evaluating with 256 clients (Figure 4(c)).

In this experiment, we also evaluate the number of minor page faults per node. This metric presents important information regarding the amount of data moved around the NUMA nodes. Minor page faults occur in two situations: (1) in the data first touch and; (2) in the remote access to data. The first touch situation means the OS will allocate memory for the thread in the local node. The remote access situation means the same data has already been touched by another thread on a remote node. The new fetch generates a new minor page fault leading to remote access with additional cost to move data around. We ran the experiments varying from 1 to 256 concurrent clients expecting performance drop after 16 to 32 active threads due to the number of cores available.

As expected, we observe that the number of page faults per second increases along with the number of concurrent clients. We also observe the increasing usage of interconnection bandwidth when all cores are let to the system. In particular, we observe that 16 concurrent users executing Q6 moves around 840 MB/s of interconnection traffic jumping to 8 GB/s with 256 concurrent users (Figure 4(c)). It is clear that more clients or more cores means more pressure: the side-effect when the OS forces to keep load balance without an auxiliary mechanism to detect a good thread mapping.

2) Impact of thread scheduling: **Impact of thread scheduling:** We investigate the thread scheduling performed by the OS with a single client execution. Figure 5 presents the migration of the threads spawn by MonetDB to execute Q6[2]. The different colors indicate different NUMA nodes and the different tones indicate the different CPU cores with the same node. This plot shows that threads migrate several times across the cores along the execution time. This showcases the effort of the OS to maintain the load balance. However, as we showed on the previous result, the traditional scheduling policy used by the OS is not NUMA-aware, which costs extra latency penalty to the cores access remote memory.

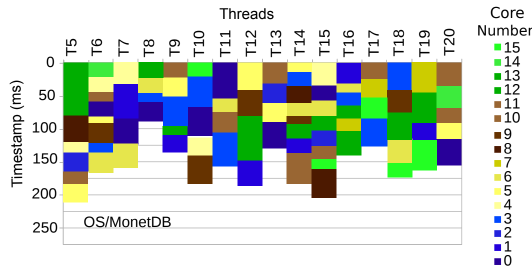


Fig. 5: Evaluating the lifespan and the migration between cores of the threads generated by Q6 in a single-client execution with all the 16 cores available.

We also observe the execution of each worker thread when running Q6 to understand the parallelism on MonetDB that leads to access to data partitions and possible data movement. Figure 6 shows the worker threads for Q6 as presented by the Tomograph facility [11]. Notice that we removed from our analyzes along the paper all the administrative threads. For instance, the first operator of the query plan in gray bar the *thetasubselect* (line 1 from Figure 3) is executed by 15 threads meaning parallel access to disjoint partitions of the $l\_quantity$ column internally implemented as a vector called Binary Association Table (BAT) in MonetDB. The parallel access to many data partitions brings pressure on the OS scheduler to find good NUMA-locality reflecting the observed numbers of cache misses and HT traffic.

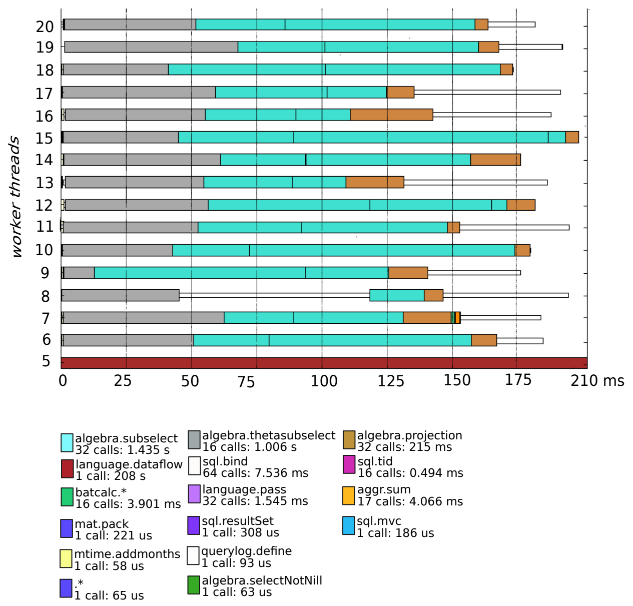


Fig. 6: A screenshot of the Tomograph facility tracking the 16 threads spawn by MonetDB to execute Q6.

III. The elastic multi-core allocation mechanism

In this section, we present our mechanism to guide the dynamic allocation of CPU cores for a given OLAP workload. We have particular interest in OLAP, because queries may provide many different CPU and memory consumption patterns in the same workload [12]. Therefore, there is constant pressure on the OS to update the allocation of the threads across the NUMA nodes for keeping load balance and data affinity.

We establish the following goals of our mechanism. First, it needs to promptly react to the fluctuations of CPU and primary memory consumptions to help the OS allocating threads with the lowest possible negative impact on performance. In this context, the mechanism should easily scale-out and dynamically find the number of cores to tackle current workload in contrast to the static manner implemented by SAP Hana [13] and Shore-MT [9]. Second, it must be straightforward to integrate to any DBMS of the user choice to motivate and leverage its usage.

A. Overview

Our mechanism implements an abstract model based on performance state transitions of the running database. This model is referred to as PetriNet Predicate/Transition (PrT) and has the ability to define performance conditions, or predicates, over monitoring information of computing resources consumed by the database threads. The predicates validate whether performance state transitions occurred. PrTs are commonly used in the literature to model performance properties of dynamic and concurrent systems [14], [15]. A PrT is an

[2]The number of worker threads is set one thread per core according to the documentation: https://www.monetdb.org/Documentation/monetdb-man-page.

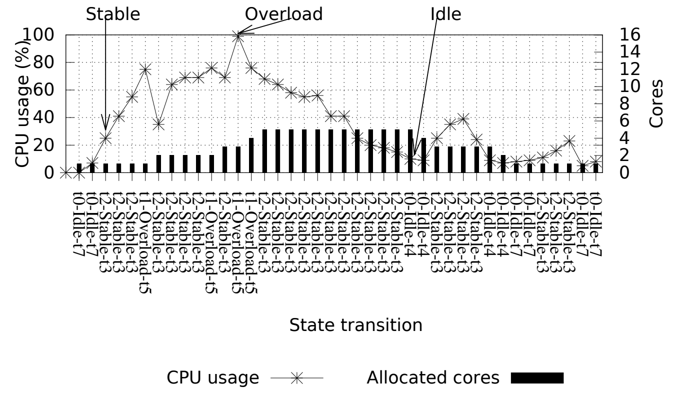


Fig. 7: State transitions of TPC-H Q6 and the allocation of cores: the X-axis depicts the transitions fired over time. The Y-axis on the left edge is the CPU usage (%). The Y-axis on the right edge is the number of allocated cores.

oriented bipartite graph representing the flow of tokens from one place downstream to another place around the net topology (i.e., no bidirected edges). A token represents an amount of a given entity. For instance, a token represents the number of allocated CPU-cores or their usage and interruptions.

We detail our model with CPU-load information for multi-core allocation to ease the understanding of the PrT, although the abstract nature of the model allows adding any other resources and metrics to improve the allocation accuracy. For instance, we can model the PrT with interconnection traffic in the integrated memory controller (HT/IMC) instead of CPU-load. Actually, in our experiments we explore the model with these two different computing resources (see Section V).

The abstract model is formally defined as a tuple with domain $\{P, T, F, R, M\}$ starting with the disjoint finite sets places and transitions $P \cap T = \emptyset$. The essential net structure is the subdomain $\{P, T, F\}$. Based on performance limits, presented in Section IV-A, we define the places $p \in P$ as performance states of a database under execution (i.e., $\{Idle, Stable, Overloaded\}$)

The transitions $t \in T$ define the conditions to switch the performance states according to the variables $v = \{u, n_{alloc}, n_{total}\}$, where u represents the average CPU-load of a NUMA node which is given in terms of percentage, $n_{alloc} \in \mathbb{N}$ the number of allocated cores and n_{total} the total number of cores available in the hardware. To illustrate our definitions, we consider the allocation of cores along the single execution of the TPC-H Q6 depicted on Figure 7. When the load of the threads goes up, the transition is promptly fired between performance states to allocate cores to the OS. Analogously, when the load goes down, cores are released.

The set F defines the arcs $<p_i, t_j>$ or $<t_j, p_i>$ between places and transitions to satisfy the condition $F \subseteq (P \times T) \cup (T \times P)$. Two functions define the flow relation: $Pre(P \times T)$ and $Post(T \times P)$. The Pre function defines the outgoing place to the incoming transition with an arc $<p_i, t_j>$ if and only if $Pre(p_i, t_j) \neq 0$. For instance, "$Stable - t_3$" means the validation of the *Stable* state at transition t_3. Analogously, the $Post$ function defines the outgoing transition to the incoming place with an arc $<t_j, p_i>$ if and only if $Post(t_j, p_i) \neq 0$. For instance, "$t_2 - Stable$" means a valid condition from transition t_2 fired the *Stable* state. Figure 8 presents the model structure as an incidence matrix $A^T = Post - Pre$.

$$\begin{array}{c} Post \\ p_0 \\ p_1 \\ p_2 \end{array} \begin{array}{c} t_0\, t_1\, t_3 \\ \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix} \end{array} - \begin{array}{c} Pre \\ p_0 \\ p_1 \\ p_2 \end{array} \begin{array}{c} t_0\, t_1\, t_3 \\ \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \end{array} = A^T \begin{array}{c} p_0\; p_1\; p_2 \\ \begin{pmatrix} -1 & 1 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 0 \end{pmatrix} \end{array}$$

Fig. 8: The transpose A^T orients the flow relation based on pre-conditions Pre and post-conditions $Post$.

Finally, the net inscription $\{R, M\}$ defines the semantics of the model, where $R : T \rightarrow <oper, bool>(X)$ is a well-defined constraining mapping, which associates each transition T with a first order logic formula defined in the underlying algebraic specification [16]. This logic defines the conditions in the network. For instance, the condition $u \geq 70$ means to fire a transition when the CPU-load overtakes 70%. M: $P \rightarrow \mathbb{N}$ is the initial marking with the initial token distribution to every place. $M(p)$: $u \rightarrow \mathbb{N}$ is a function to tell how many tokens reside in that place. For instance, this function tells how many cores are available.

B. The abstract model specification

In this subsection, we specify the rule-condition-pipeline and the definitions to build our PetriNet model. Initially, we define the set of places $P = \{Stable, Idle, Overload, Provision, Checks\}$ and transitions $T = \{t_0, \ldots, t_7\}$. The first three places are related to the database performance states and the last two are complementary places to assess the load, where *Checks* is synchronously updated with the current resource usage and *Provision* informs the number of cores in usage. In order to facilitate understanding, we partitioned the PetriNet in sub-nets around the places *Stable*, *Idle* and *Overload*.

To decide whether to fire a transition, we set performance thresholds based on resource usage (given in percentage): th_{min} is the minimum threshold and th_{max} is the maximum threshold. If the performance thresholds are respected, the database is considered *Stable* and only monitoring is required. Otherwise, it needs to fire transitions to bring the database back to the *Stable* state.

Finally, we set the starting marks of the set M at each state. These marks define the net execution with the acceptable range of resource usage at each place, as follows: $m_0(Checks) = \{0, \ldots, 100\}$, $m_0(Idle) = m_0(Stable) = m_0(Overload) = \{0\}$, and $m_0(provision) = \{r\}$ (i.e., $a_alloc == 1$ default number of cores initially allocated). We illustrate our specification through a workload currently using 3 out of 16 cores ($n_{alloc} == 3$ and $n_{total} == 16$) of our AMD Opteron machine and CPU usage thresholds (given in load percentage) based on the rules of thumb from the literature [17] and adjustments by empirical experiments. The thresholds are defined as $th_{min} = 10$ and $th_{max} = 70$.

1) **The *overload* sub-net:** The *overload* sub-net models high CPU load when more processing cores are required to be allocated. The overloaded state is somehow expected for systems with high OLAP throughput. Instead of the common practice of letting the OS dealing with all the available CPU cores, the cores are made available to the OS on demand. The final goal is to provide cores accordingly to the workload behavior, considering the load and also the data share among the threads. For instance, Figure 9 depicts $t1$ firing when the CPU load is $u = 99\%$ with th_{max}=70%, considering 3 cores provisioned out of 16 cores. Next, $t5$ fires the transition to *Provision* to allocate one more core if there are still cores available to be allocated (i.e., $n_{alloc} < n_{total},\ n_{total} == 16$). Figure 7 illustrates this transition as "$T1 - Overload - T5$", during Q6 execution. In parallel, $t5$ fires *Checks* to validate the CPU load, that goes down to $u = 68\%$.

In this example, the incidence matrix of the *Overhead* sub-net represents the fired arcs of connections in the pre-conditions, as $\{Checks{-}t_1, Overload{-}t_5, Provision{-}t_1)\}$. Notice that the arc "$Overload{-}t_6$" is not set in the *Pre* matrix, because the validation does not allow any post-condition for t_6, since there are still NUMA cores available. The post-conditions are $\{t_5{-}Checks, t_5{-}Provision, t_1{-}Overload)\}$. The incidence matrix is presented, as follows:

$$\begin{array}{l} Post \\ Checks \\ Over \\ Provision \end{array} \begin{array}{c} t_1\ t_5 \\ \begin{pmatrix} 0 & u \\ n_a & 0 \\ 0 & n_a \end{pmatrix} \end{array} - \begin{array}{l} Pre \\ Checks \\ Over. \\ Prov. \end{array} \begin{array}{c} t_1\ t_5 \\ \begin{pmatrix} u & 0 \\ 0 & n_a \\ n_a & 0 \end{pmatrix} \end{array} = A^T \begin{pmatrix} \text{Checks} & \text{Over.} & \text{Prov.} \\ -u & n_a & -n_a \\ u & -n_a & n_a \end{pmatrix}$$

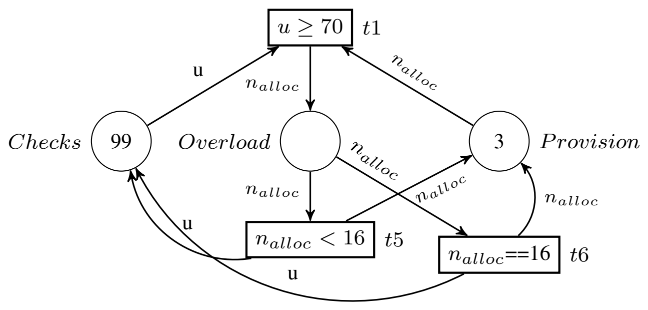


Fig. 9: Tokensthrough transition in the Overload sub-net to allocate one core with $u = 99\%$, $n_{alloc} = 3$ out of $n_{total} = 16$ cores and $th_{max} = 70\%$.

2) **The *idle* sub-net:** The *Idle* sub-net models the database in idle state when cores can be released. The condition to fire t_0 is set to low CPU usage across multiple cores. Next, t_4 fires if there are still cores to be released (i.e., $n_{alloc} > 1$), otherwise t_7 fires the transition to *Checks* waiting further updates in the variables. Transition $t7$ bounds the least number of CPU in the system. For instance, Figure 10 depicts t_0 firing when the CPU load is $u = 10\%$ with $th_{min} = 10\%$ for 5 cores provisioned. Next, t_4 fires the transition to *Provision* to release one core ($n_{alloc} = 4$). Figure 7 depicts this transition, as "$t_0{-}Idle{-}t_4$". Concurrently, t_4 fires to *Checks* to validate the load again that goes down to 5% after the transition.

The incidence matrix of the *Idle* sub-net, according to the example, depicts the fired arcs of connections in the pre-conditions, as $\{Checks - t_0, Provision - t_0, Idle - t_4)\}$. Notice the arc "$Idle - t_7$" is not set in the *Pre* matrix, because the validation does not allow any post-condition for t_7. Therefore, the fired arcs of connections in the post-conditions, are $\{t_4 - Checks, t_4 - Provision, t_0 - Idle)\}$. The incidence matrix is presented, as follows:

$$\begin{array}{l} Post \\ Checks \\ Idle \\ Prov. \end{array} \begin{array}{c} t_0\ t_4 \\ \begin{pmatrix} 0 & u \\ n_a & 0 \\ 0 & n_a \end{pmatrix} \end{array} - \begin{array}{l} Pre \\ Checks \\ Idle \\ Prov. \end{array} \begin{array}{c} t_0\ t_4 \\ \begin{pmatrix} u & 0 \\ 0 & n_a \\ n_a & 0 \end{pmatrix} \end{array} = A^T \begin{pmatrix} \text{Checks} & \text{Idle} & \text{Prov.} \\ -u & n_a & -n_a \\ u & -n_a & n_a \end{pmatrix}$$

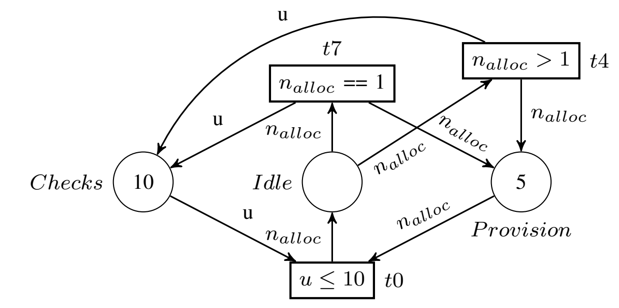


Fig. 10: Tokens through transition in the Idle sub-net to release one of the 5 CPU cores with $u = 8\%$ and th_{min} = 10%.

$$\begin{array}{l} Post \\ Checks \\ Stable \end{array} \begin{array}{c} t_2\, t_3 \\ \begin{pmatrix} 0 & u \\ u & 0 \end{pmatrix} \end{array} - \begin{array}{l} Pre \\ Checks \\ Stable \end{array} \begin{array}{c} t_2\, t_3 \\ \begin{pmatrix} u & 0 \\ 0 & u \end{pmatrix} \end{array} = A^T \begin{pmatrix} \text{Checks} & \text{Stable} \\ -u & u \\ u & -u \end{pmatrix}$$

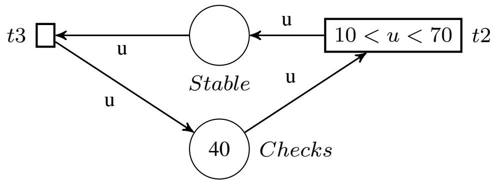


Fig. 11: Tokens through transition in the Stable sub-net with $u = 40\%$ and threshold between 10% and 70%.

3) **The *stable* sub-net:** The *Stable* sub-net models the database in stable performance state, when the current number of cores is sufficient for the current load. This state is represented by the combined load usage u of the running cores kept between the thresholds. The *Pre* matrix for the $Stable$ sub-net presents the arcs of connections between "$Checks - t_2$" and "$Stable - t_3$" (i.e., pre conditions). For instance, Figure 11 depicts this sub-net firing t_2 for a given CPU load of $u = 40\%$ with $th_{min} = 10$ and $th_{max} = 70$.

The *Post* matrix shows the arcs of connections between "$t_2 - Stable$" and "$t_3 - Checks$" (i.e., pos condition). The incidence matrix for the stable sub-net shows the effect of the t_2 and t_3 firing. The first line presents that firing removes the token from $Checks$ and adds to $Stable$. The second line shows t_3 that firing removes the token from $Stable$ and adds to $Checks$. No core allocation is required in this scenario. Figure 7 depicts this transition, as "$t_2 - Stable - t_3$". The incidence matrix is presented, as follows:

IV. The allocation of processing cores

Our optimization goal is to find out the number of cores that best accommodate the current OLAP load and prevent both under-utilization or over-utilization of the multi-cores. Moreover, our mechanism must take into account the data allocation across the NUMA regions allocating cores to specific nodes (near the address space of the first touch), avoiding data movement and thread migrations.

In this section we describe the core allocation performed by our mechanism, called *Adaptive Priority*, and we also discuss two other simpler allocation policies *Sparse* and *Dense*. We also show the implementation details to keep the performance counters up-to-date in the abstract model.

A. The local optimum number of cores

The Local Optimum Number of Cores (LONC) is achieved when the CPU-load is within the *Stable* sub-net. We refer as to "Local Optimum", because we only take into account the arithmetic CPU-load average of the active database threads.

Formally, we define the LONC, as follows:

$$\forall\ w\ \exists\ n_{alloc} | (th_{min} < u < th_{max}) \land p(n_{alloc}) \geq p(n_{total}) \quad (1)$$

where:

- w is the current workload of the database threads;
- u is the average resource usage of database threads;
- n_{alloc} is the number of allocated CPU cores ($n_{alloc} \leq n_{total}$);
- n_{total} is the number of available CPU cores;
- $p(x)$ is the performance function, where x assumes the number of CPU cores n_{alloc} or n_{total};

To any OLAP workload w, there is a certain number of CPU cores n_{alloc} such that the load of each core are between the minimum and maximum *thresholds*, in which the database performance $p(n_{alloc})$ is equal or better than the performance $p(n_{total})$ with all the CPU cores available in the hardware. The performance function $p(x)$ relies on system counters provided by the OS and the DBMS.

In our implementation, the performance function uses the OS's functionalities to keep the counters up-to-date in the abstract model and improve the accuracy of the resource allocation. Initially, we use the *cgroups*, which is a kernel feature, to isolate the threads of the DBMS, and their future children, into specific hierarchical groups (our allocation modes). With *cgroups*, we gather the process identification numbers (PID) of the threads to monitor their execution and limit their available resources (e.g., cores). We implemented a priority queue with the PID of the threads and their NUMA resource usage information (e.g., their execution core and address space). We detail the algorithm to maintain the priority queue later in this section. We also use the *MPstat* for CPU load information of the PIDs and we use the *Likwid* [18] tool to get the L3 cache data misses (*L3CACHE* Likwid option), the bandwidth on the NUMA interconnection links (HT at *NUMA_0_3* Likwid option) and the memory traffic (*MEM* options on Likwid) [3].

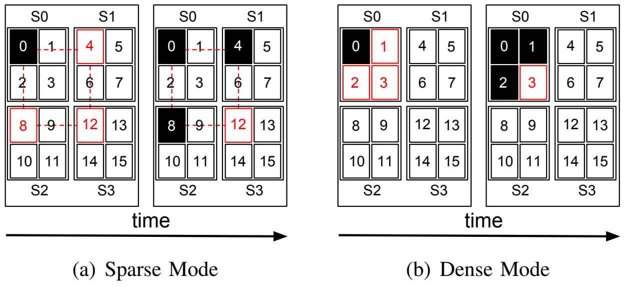


Fig. 12: The Sparse and Dense modes over time. Only the black boxes (i.e., cores) can be accessed by the OS. The red boxes are the next cores to allocate.

The CPU load for the LONC calculation also considers other processes of the DBMS besides the threads from query execution. When operations fully utilize the CPU ($u = 100\%$), a new core is allocated to maintain stable use and prevent DBMS threads to stall. The expected result is to achieve better response time with more cache hits and less remote memory access compared to the current allocation of CPU-cores for OLAP. Following our example, Figure 7 depicts the execution of Q6 supported by our mechanism. This figure exemplify the mechanism behavior, where cores allocated when the load goes up to 99% (transition $t1 \rightarrow t5$) and released when the load goes down to 8% (transition $t0 \rightarrow t4$).

B. The multi-core allocation modes

In this section, we present the core allocation modes to address the needs of different DBMS thread models. The *thread model* may present more shared memory between threads and require to allocate them in the same node. Instead, the *process model* may not present much memory shared among the threads. In this case, threads are allocated in different nodes to avoid memory competition. To illustrate these allocation modes, we consider the NUMA machine depicted in Figure 2.

1) **The *sparse* and *dense* modes:** The *Sparse* and *Dense* modes performs dynamic core allocation following simple rules during the workload execution. The definition of the general allocation mode function is straightforward. It maps a NUMA node with index i to its j^{th} core and is defined by $core(i, j) = d.i + j$, where $1 \leq j \leq d$. In our function, d is a constant to represent a d-ary node machine, shown as $d = 4$ by Fig. 12 to represent our four-node AMD Opteron machine.

The *Sparse* mode iterates over i, j to allocate one core at a time in a different NUMA node. The *Dense* mode iterates over j, i to allocate one core at time in the same NUMA node. Fig. 12 shows the sparse and dense mode allocation over time.

[3]The likwid monitoring options for the ADM Opteron is available at: https://github.com/RRZE-HPC/likwid/tree/master/groups/k10

2) **The *adaptive priority* mode:** The goal of our adaptive multi-core allocation mode is to spot the next allocation NUMA node based on memory usage information. A priority queue is used to indicate the node with the largest/smallest amount of allocated memory (on top/bottom priority) and the model allocates/releases a core near to such address space. Each entry of the priority queue keeps the PIDs of the active threads with their address spaces and the number of pages per NUMA node.

In the workflow of the adaptive mode, the number of pages per NUMA node is recorded in a counter to acknowledge the node with the biggest amount of pages. The NUMA node with highest priority is the one with the biggest counter value. Analogously, the NUMA node with the smallest counter value is the one with lowest priority. If a new CPU-core is needed, it is allocated in the highest priority NUMA node. If a CPU-core needs to be released, a core in the NUMA node with the lowest priority is deallocated.

V. Experimental analysis

We implemented our prototype as an application program on top of Debian Linux 8 ("Jessie" with kernel 3.16.0-4-amd64) and we compare our mechanism to the OS scheduling of threads spawn by the MonetDB (v11.25.5) DBMS. We also evaluate the mechanism with the NUMA-aware DBMS SQL Server (v2017 Developer RC2) with column store index in Section V-C as both DBMS implement similar thread model based on Volcano and implement similar parallel access to column data structures (i.e., BAT or vectors) [6]. We performed the experiments on a NUMA machine (previous illustrated on figure 2) formed by 4-node with a Quad-Core AMD Opteron 8387 each, executing at 2.8 GHz. Each Opteron socket is formed by four cores with private L1 cache (64 KB) and L2 cache (512 KB) and a shared L3 cache (6 MB), with the NUMA nodes interconnected by Hyper-Transport (HT) link 3.x achieving 41.6 GB/s maximum aggregate bandwidth (32-bit). This machine includes 64 GB of DDR-2 main memory and 1.8 TB SATA disk. We let all the 16 cores available to the DBMSs when running without the support of our mechanism. We present the results considering the average over 10 executions for each allocation mode.

Inside the mechanism, we coded the thresholds to $th_{min} = 10$ and $th_{max} = 70$ following the rules of thumb in the literature [17] and these values are kept in all the experiments. Nevertheless, such parameters could be used to define trade offs between core utilization and performance, we experimented different thresholds, but decreasing th_{min} lets too many cores in idle state, while increasing th_{max} leads to contention with too many busy cores. We measured the overhead of our mechanism in terms of the flow of tokens in a 5×8 matrix to trigger a transition in our abstract model. In the dense mode the flow of tokens takes on average *0.017 seconds*, while the sparse mode takes *0.021 seconds* and the adaptive mode *0.031 seconds*. The CPU load average computing the state transition is less than *1%*. Notice that we require only a single instance of our mechanism to support all DBMS clients.

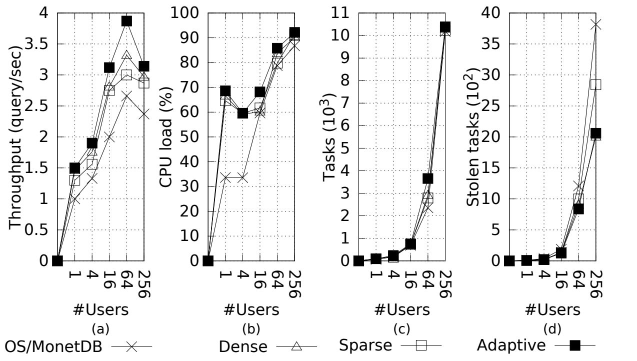


Fig. 13: Performance metrics when processing an increasing numbers of concurrent clients running the *thetasubselect*.

A. Analysis of the TPC-H Q6

In this section, we investigate the impact of our mechanism (*Adaptive*), the normal OS scheduling (*OS/MonetDB*), and two static modes also supported by our mechanism (*Sparse* and *Dense*). We focus our analysis on the *thetasubselect* operator, because it moves large amounts of data prior to other operators in the plan, which is the $l\_quantity$ column with uniform distribution and 45% of selectivity (see Section II-B). This scenario benefits the dense mode (and the adaptive), because the OS scheduler is expected to map threads close to the memory address space of the column.

1) **Impact of scheduling:** In this section, we present the impact of the allocation modes on thread scheduling as we increase the concurrency, that is number of clients in parallel. We performed the same execution protocol described by [13], but limited to 256 concurrent users running the modified version of Q6 in 1 GB (size of raw data).

Figure 13 shows NUMA-related performance metrics. The query throughput metric showed its peak at 64 concurrent users, but the performance improvements happened in all scales (Figure 13 (a)). Overall, the adaptive presented higher efficiency and performance, achieving 25% more query throughput than the OS scheduler. The metrics of CPU load and tasks remained similar in all the modes (Figures 13 (b) and (c) respectively). However, the OS effort to keep load balance resulted in 46% more stolen tasks than our adaptive mode (Figure 13 (d)).

Figure 14 shows memory usage in order to understand the impact of the stolen tasks. The results show the difficulty of the OS scheduler to find thread/data locality. We observed that our mechanism (adaptive) decreased the L3 cache misses in 43% improving memory throughput by 27% (Figures 14 (a) and (b)). With poor thread mapping, the OS scheduler achieved the highest interconnection usage (HT Traffic) among all the experiments meaning high data movement (Figure 14 (c)). With less data movement, the adaptive mode exploited better the memory bandwidth of all sockets simultaneously matching the memory throughput behavior observed in NUMA-aware DBMS: SQL Server (see Sec. V-C) and SAP Hana (see [13]). Considering the simple dense and sparse modes, the dense mode let underused the last socket S_3, while the sparse mode showed more interconnection traffic than the other modes and consequently lower memory throughput. This result shows the impact of each allocation mode: nearby (dense) and far apart (sparse). Nevertheless, both modes presented less stolen tasks and better memory throughput than the OS scheduler.

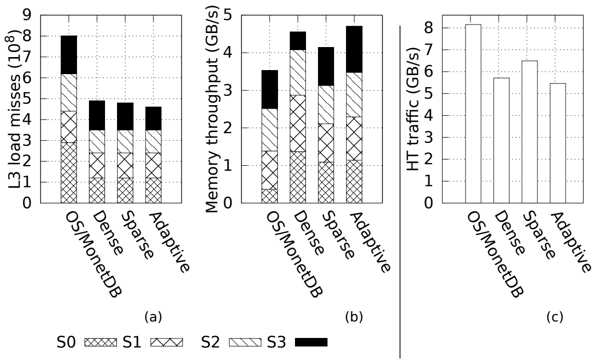


Fig. 14: Memory access metrics with 256 clients running the *thetasubselect*.

2) **Impact of selectivity:** In this section, we present the impact of different allocation modes when 256 concurrent clients fetch different amounts of data in 1 GB database. This means measuring the impact of memory-intensive column scans in different levels of selectivity.

Figure 15 depicts the results of L3 cache misses. The lower the value, the better is the memory usage, which is observed in all our allocation modes when compared to the OS scheduler. Figure 15 (a) shows that the cache size is insufficient to keep the materialization of Q6 with more than 64% of selectivity with a spike in L3 load misses. Our allocation modes (adaptive, dense and sparse), present on Figures 15 (b), (c) and (d) respectively, offset the cache insufficiency with better memory throughput. Interestingly, none of our modes at any selectivity, even retrieving 100% of the column, presented more L3 cache misses than the OS scheduler when retrieving more than two-thirds of the column (i.e., 64% of selectivity).

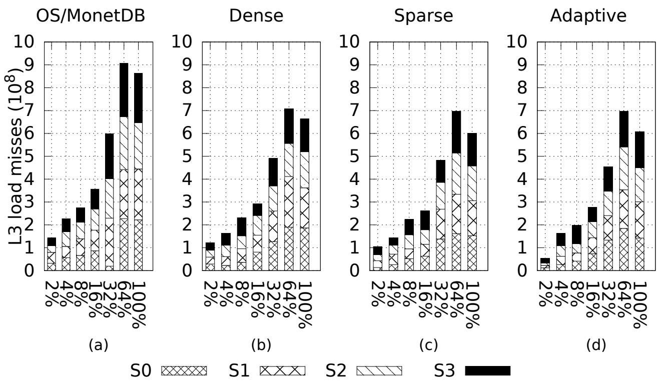


Fig. 15: Evaluating L3 cache misses with different selectivities and 256 clients processing the *thetasubselect*.

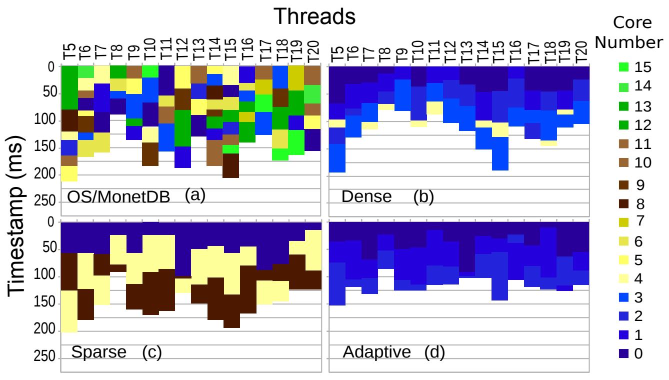


Fig. 16: Lifespan and the migration of cores of each thread of the original TPC-H Q6 with a single client.

3) *Impact of the Migration of Threads:* In this section, we compare the migration of threads during their lifetime and in which cores they executed. Here, we evaluated the complete query plan of the Q6 statement to study the impact of the migration of threads to access the materialized data in different nodes along the query pipeline. We ran this experiment with 1 client in 1 GB database to compare to the results of Section II.

As expected, the dense and adaptive modes made the scheduler run the threads in the same NUMA node most of the time (Figs. 16 (b) and (d)), while the OS scheduler mapped MonetDB's threads all over the nodes (Fig. 16 (a)). Interestingly, the threads migrated several times from one core to another and sometimes they even returned to the same core. This shows that the current OS scheduling is not NUMA-friendly to the database load and is constantly migrating threads to find NUMA-affinity. The sparse mode is the closest to the expected behaviour of the OS scheduling assigning threads far-apart, however, Fig. 16 (c) shows less thread migration when gradually offering less cores to the OS.

Next, we analyze the performance metrics when our mechanism exposes only the efficient subset of cores to the OS scheduler. Figure 17 (a) shows that the response time of Q6 was 27% faster with the adaptive mode. Figures 17 (b) and (d) respectively show 9× more HT traffic and 2× more L3 cache misses for the OS scheduler than for our adaptive mode, impacting negatively in the performance and energy consumption. With less cores available for mapping threads, our mechanism helped the OS to take good scheduling decisions resulting in higher cache hits and less remote accesses.

B. PetriNet with HT/IMC State transition

In this experiment, we switch the state transition strategy of the PrT from CPU-load to the ratio of traffic in the integrated memory controller (HT/IMC). Our goal is to show the flexibility of our model, that can fit different strategies and metrics to take decisions. The HT/IMC ratio defines how NUMA-friendly is the system: the system is able to process more data with less interconnection traffic. Thus, we can observe if the interconnect usage achieves a better allocation strategy than CPU load. We empirically set the HT/IMC ratio

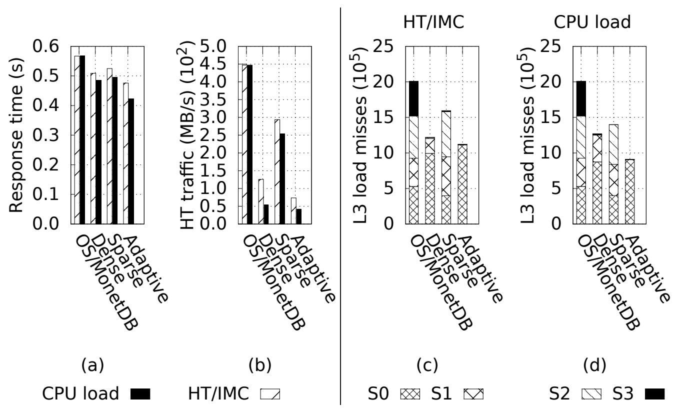


Fig. 17: Performance metrics of Q6 with 1 client in 1 GB database running our model with two different state transition setups: CPU load and HT/IMC.

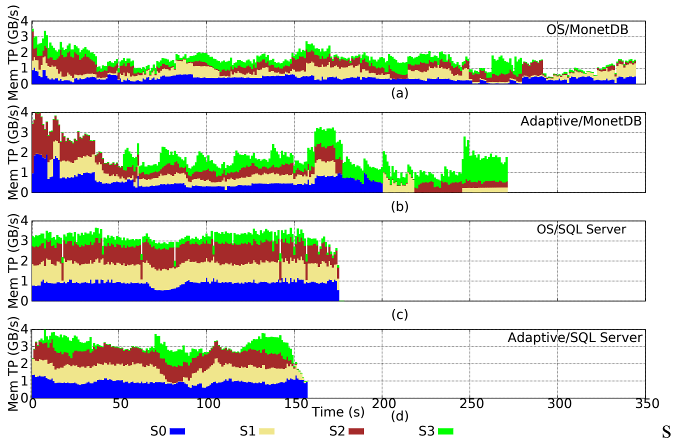


Fig. 18: Stable phases workload with 256 concurrent clients in 1 GB database.

thresholds to $th_{min} = 0.1$ and $th_{max} = 0.4$ based on the best achieved results. We extract the HT/IMC ratio of individual queries from their PIDs in the priority queue.

Figure 17 compares side-by-side the PrTs configured with both state transition strategies. Overall, we observe similar behavior in the migration of the threads with slightly different results in response time, HT traffic and L3 misses, Figs. 17 (a), (b) and (d) respectively. In some cases the CPU load presented half of the HT traffic. With the HT/IMC metric, the OS started filling in the L3 cache, but when the mechanism chooses to allocate a new core far apart all data loaded into L3 is lost, generating more cache misses and increasing the execution time (Fig. 17 (c)). Therefore, the PetriNet with HT/IMC state transition takes longer to make the allocation with impact in thread movements.

C. TPC-H Benchmark

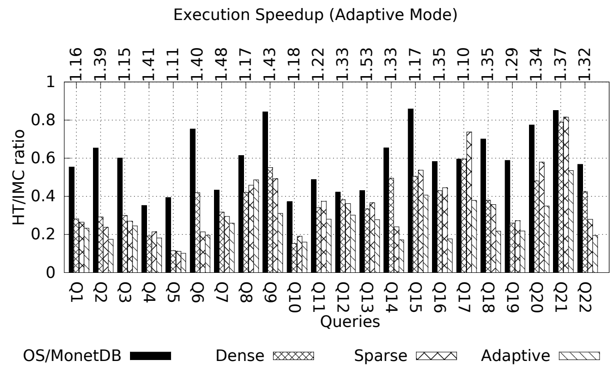
In this section, we analyze the impact of our mechanism running the entire TPC-H query stream using two workloads. Moreover, we analyze our mechanism with MonetDB and the NUMA-aware DBMS SQL Server. We do not aim to make a comparison between DBMSs. Instead, our goal is to show that our mechanism is orthogonal to their thread and data placement strategies improving their NUMA-affinity with the elastic allocation of cores that assists the OS scheduler.

1) Stable phases workload: **Stable phases workload:** In this section, we present the impact of our mechanism with a more dynamic data access pattern. We present the results for the concurrent execution of all the TPC-H queries to show that our address mapping scheme self-adapts to the changing workload and keeps the processing in specific NUMA nodes. The workload is divided in phases, where each phase is the concurrent execution of each query at a time by 256 users in 1 GB database. Figure 18 shows the results for memory throughput (y-axis) and execution time (x-axis).

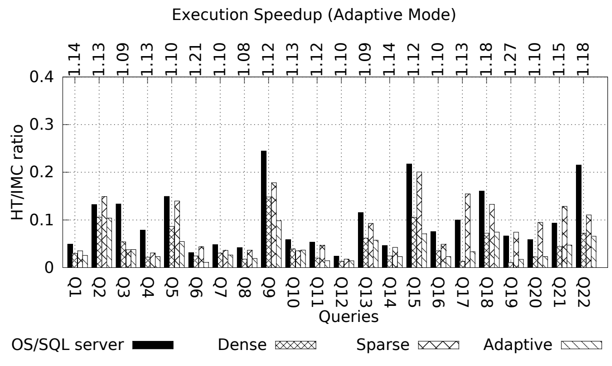
Using MonetDB, the adaptive mode improved the OS scheduling task of finding thread-data affinity. It was 41% faster with higher memory throughput than the OS/MonetDB. For instance, the execution until the 50^{th} second shows that our scheme focused on sockets S_0 and S_2, while between the 200^{th} and 220^{th} seconds it switched to sockets S_1 and S_3. Besides, we observe that the OS is oblivious to the database load without our mechanism and constantly uses the cores on socket S_0 over the whole execution (Figs. 18 (a) and (b)).

When executing with SQL Server, we observed that a NUMA-aware DBMS better exploit the sockets simultaneously, and even so, our mechanism decreased response time. SQL Server associates threads and cores to improve affinity, but with less cores available, for instance between the 40^{th} and 60^{th} seconds, there is less effort to maintain coherence of such association (Figs. 18 (c) and (d)).

2) **Mixed phases workload:** Here, we present the results executing 256 concurrent users continuously running a random query out of the 22 queries of TPC-H. Figures 19(a) and 19(b) show the speedup split per query of our mechanism compared to the OS scheduler with MonetDB and SQL Server, respectively, and the proportion of the interconnection traffic in relation to the traffic of the HT/IMC as described in [9]. The adaptive mode presented the best per query speedup, achieving up to 1.48× and up to 4× smaller per query HT/IMC ratios than the OS scheduler with MonetDB. For SQL Server the speedup was as higher as 1.27× achieving 4× smaller HT/IMC ratio. In MonetDB, we observed 3× smaller HT/IMC ratios for queries Q8 and Q9 (in SQL Server 2× smaller). These queries implement the largest number of join operations and present a high degree of parallelism. The result shows that the threads spanned from Q8 and Q9 are finding more data in local memory compared to the DBMSs without the adaptive mode. We conclude the same for queries Q19 and Q22 with almost 3× smaller HT/IMC ratios in both MonetDB and SQL Server, because we reduce the number of cores where threads process "IN" predicates (a series of constant values shared in a list). The decrease in the interconnection usage is a direct reflect of the adaptive mode: cores allocated on nodes with more accessed pages and cores released with the least number of accessed pages. Therefore, threads require less remote accesses and migrations, increasing the efficiency to



(a) MonetDB



(b) SQL Server

Fig. 19: Mixed phases workload split per query with 256 concurrent clients in 1 GB database. On the y-axis, the HT/IMC ratio shows how NUMA-friendly is the system (the smaller, the better). Topmost is the performance speedup for adaptive mode.

find data affinity.

3) Energy evaluation: Now, we analyze the effect of our mechanism on the energy consumption. We performed estimates of energy consumption considering our best policy (adaptive) to the traditional OS scheduler with MonetDB. We used the values regarding the Average CPU Power (ACP) for the processor used in our experiments. We also obtained the average energy per bit transferred for HT [19]. Thus, we performed estimates for all the benchmark executed on previous experiment using the results from hardware counters.

Figure 20 shows the energy consumption given in Joules consumed during the execution of each one of the 22 queries (split between CPU and HT). We notice that most of CPU energy savings came from the smaller execution time, while HT energy savings came from the reduced number of data transfers. We can observe minimum savings of 9.67% for CPU (Q17) and minimum savings of 46.22% for HT (Q12). Analyzing the geometric mean, we saved 22.93% for CPU and 63.20% for HT, across the queries leading to a total energy saving of 26.05% on the system consumption. All in all, we observe that our mechanism improved the scheduling of database threads with direct reduction on data transfers between the NUMA nodes leading to better energy consumption.

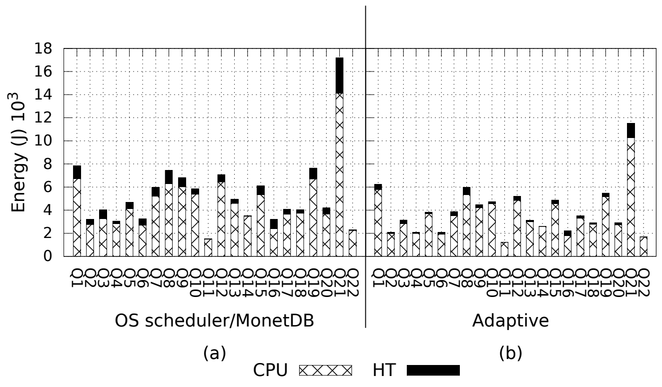


Fig. 20: Energy estimations for CPU and HT, executing the 1 GB TPC-H queries with 256 concurrent clients in MonetDB.

VI. Related work

The impact of the NUMA effect motivated several recent work on query processing focusing either on particular query operators or on parallel execution frameworks. Our paper is related to those improving the resource allocation for the parallel execution frameworks, but we refer to [2], [20] for reading on NUMA-aware query operators.

A number of papers focus on thread/data placement strategies with some differences on the allocation of NUMA cores. The Shore-MT storage manager presents static "hardware islands" for processing OLTP in pinned threads [9]. Different from our mechanism, the "hardware islands" do not scale-out and the optimum size of the island is yet undetermined. Likewise, SAP Hana and the ERIS storage engine also allocate NUMA cores statically at the startup [13], [21]. As observed in our paper, the static allocation of cores may lead to inefficient thread mapping when many different access patterns of stable and mixed phases workloads hit the system.

The scheduler of HyPer [2] and DB2 BLU [22] assume explicit control over the dispatching of query fragments, called "morsels". In this model, the parallelism degree can change at runtime to execute "morsels" in local nodes opposed to Volcano where parallelism is statically set at planning time with identical pipelined segments of the query plan. The distribution of morsels occurs to threads statically pinned to the cores upon the location of the shared data structures (input/output buffers) to avoid moving threads and loose NUMA-locality. For instance, threads run the grouping operation in two phases to avoid remote access: (1) locally building hash tables and (2) picking partitions to merge the corresponding entries remotely. However, none of these work clearly focus on the elasticity of cores needed by OLAP and the number of threads are statically pinned to the available cores. Instead, our mechanism presents an orthogonal approach that can deliver to morsels a dynamic sub-set of cores on top of a priority queue to efficiently adapt to OLAP workloads. In addition, different priority policies can be implemented as needed due to the abstract nature of

PetriNets, as shown with CPU load and HT/IMC ratio.

Recent work on Volcano-style schedulers also assume the static allocation of cores and threads, which usually uses inaccurate memory estimation [3]. In Oracle the individual operators are unaware of parallelism [2], which leads to inefficient thread placement and eventually increases data movement, although an adaptive data distribution scheme is presented in [23]. A recent prototype of SAP Hana implements an adaptive algorithm to decide between data placement and thread stealing when load imbalance is detected [5] (Li et al. [20] presented a similar tradeoff strategy). However, the number of threads changes based on the core utilization opposed to MonetDB [24] and SQL Server [6] that bound the number of worker threads and maximum data partitions to the number of cores available per socket. MonetDB and SQL Server implement similar vector structures internally (i.e., BAT) to boost parallel access to disjoint partitions of columns (discussed in Section II). However, they differ on the thread placement strategy, as observed in our results. MonetDB let to the OS the thread scheduling responsibility, while SQL Server is NUMA-aware associating threads and processors to improve affinity. All in all, the thread stealing and interconnection traffic are more compelling with dynamic partitioning when all the cores are available to the system, as shown in our results, which highlights the potential of the elastic core allocation provided by our mechanism.

VII. Conclusions

In this paper, we presented an elastic multi-core allocation mechanism for database systems implemented on top of an abstract model. Our hypothesis was that we mitigate the data movement if we only hand out to the OS the local optimum number of cores in specific NUMA nodes. Indeed, our mechanism is able to improve OLAP performance by keeping track of the performance states of the DBMS on top of monitoring facilities. Results showed performance improvements when our mechanism offered to OS only the local optimum processing cores, instead the traditional approach of making all the cores visible to the OS all the time.

Our adaptive multi-core allocation algorithm improved the OS scheduling accuracy when the cores were gradually made available. Our results showed less thread migrations with less available cores and also less remote memory accesses compared to the traditional OS scheduler with MonetDB. We exploit the potential of implementing allocation algorithms in the abstract level exploring the internal computing resource information from the OS and DBMS. Therefore, we observed an important speedup of up to 1.53$\times$ and up to 3.87$\times$ smaller per query HT/IMC ratios when compared to the traditional OS scheduler. Our estimates show that such reduction on the number of remote accesses lead to a total energy saving of 26.05% on the NUMA system. When evaluating our mechanism with the NUMA-aware SQL Server, we observed speedup of up to 1.27$\times$ and up to 3.70$\times$ smaller per query HT/IMC ratios. This highlights the need for the adaptive core allocation model to improve NUMA-affinity.

As future work, we plan to improve our model to seek for the local optimum number of cores with respect to query predicates and also evaluate the benefits of our strategy in the cloud computing context when accessing cores as needed, like meeting service level agreements (e.g., energy or data traffic). Besides, we also plan to study extensions to DBMS schedulers to take benefit from under-utilized cores to concurrent applications (e.g., mixed OLAP/OLTP).

References

[1] G. Graefe, "Encapsulation of parallelism in the volcano query processing system," in *SIGMOD.*, 1990, pp. 102–111.

[2] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann, "Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age," in *SIGMOD*, 2014, pp. 743–754.

[3] P. Larson, C. Clinciu, C. Fraser, E. N. Hanson, M. Mokhtar, M. Nowakiewicz, V. Papadimos, S. L. Price, S. Rangarajan, R. Rusanu, and M. Saubhasik, "Enhancements to SQL server column stores," in *SIGMOD*, 2013, pp. 1159–1168.

[4] J. M. Hellerstein, M. Stonebraker, and J. Hamilton, "Architecture of a database system," *Found. Trends databases*, no. 2, 2007.

[5] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki, "Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores," *PVLDB*, no. 2, 2016.

[6] P. Larson, E. N. Hanson, and S. L. Price, "Columnar storage in SQL server 2012," *IEEE Data Eng. Bull.*, vol. 35, no. 1, pp. 15–20, 2012.

[7] P. Boncz, T. Neumann, and O. Erling, "TPC-H analyzed: Hidden messages and lessons learned from an influential benchmark," in *TPCTC*, 2014.

[8] C. Lameter, "NUMA (non-uniform memory access): An overview," *ACM Queue*, no. 7, 2013.

[9] D. Porobic, I. Pandis, M. Branco, P. Tözün, and A. Ailamaki, "OLTP on hardware islands," *PVLDB*, no. 11, 2012.

[10] S. Dominico, E. C. de Almeida, and J. A. Meira, "A petrinet mechanism for OLAP in NUMA," in *DaMoN*, 2017.

[11] M. Gawade and M. L. Kersten, "Tomograph: highlighting query parallelism in a multi-core system," in *DBTest*, 2013, pp. 3:1–3:6.

[12] ——, "Adaptive query parallelization in multi-core column stores," in *EDBT*, 2016.

[13] I. Psaroudakis, T. Scheuer, N. May, A. Sellami, and A. Ailamaki, "Scaling up concurrent main-memory column-store scans: Towards adaptive NUMA-aware data and task placement," *PVLDB*, no. 12, 2015.

[14] X. He, "A formal definition of hierarchical predicate transition nets," in *Application and Theory of Petri Nets*, 1996.

[15] J. Desel and W. Reisig, "The concepts of petri nets," *Software and System Modeling*, no. 2, 2015.

[16] H. Yu, X. He, Y. Deng, and L. Mo, "A formal method for analyzing software architecture models in SAM," in *COMPSAC*, 2002.

[17] U. F. Minhas, R. Liu, A. Aboulnaga, K. Salem, J. Ng, and S. Robertson, "Elastic scale-out for partition-based database systems," ser. ICDEW12.

[18] J. Treibig, G. Hager, and G. Wellein, "Likwid: A lightweight performance-oriented tool suite for x86 multicore environments," in *PSTI*, 2010.

[19] Q. Wang and B. C. Lee, "Modeling communication costs in blade servers," in *HotPower*, 2015.

[20] Y. Li, I. Pandis, R. Mueller, V. Raman, and G. M. Lohman, "NUMA-aware algorithms: the case of data shuffling." in *CIDR*, 2013.

[21] T. Kissinger, T. Kiefer, B. Schlegel, D. Habich, D. Molka, and W. Lehner, "ERIS: A NUMA-aware in-memory storage engine for analytical workload," in *ADMS*, 2014.

[22] V. Raman, G. K. Attaluri, R. Barber, N. Chainani, D. Kalmuk, V. KulandaiSamy, J. Leenstra, S. Lightstone, S. Liu, G. M. Lohman, T. Malkemus, R. Müller, I. Pandis, B. Schiefer, D. Sharpe, R. Sidle, A. J. Storm, and L. Zhang, "DB2 with BLU acceleration: So much more than just a column store," *PVLDB*, vol. 6, no. 11, pp. 1080–1091, 2013.

[23] S. Bellamkonda, H. Li, U. Jagtap, Y. Zhu, V. Liang, and T. Cruanes, "Adaptive and big data scale parallel execution in oracle," *PVLDB*, vol. 6, no. 11, pp. 1102–1113, 2013.

[24] M. Gawade and M. L. Kersten, "NUMA obliviousness through memory mapping," in *DaMoN*, 2015.