Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

M.L. Kersten, S. Shair-Ali, C.A. van den Berg

Performance analysis of a dynamic query processing scheme

Computer Science/Department of Algorithmics & Architecture Report CS-R9050 September

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Research (N.W.O.).

Performance Analysis of a Dynamic Query Processing Scheme *

M.L. Kersten
S. Shair-Ali
C.A. van den Berg

*Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam
The Netherlands*

Abstract

Traditional query optimizers produce a fixed query evaluation plan based on assumptions about data distribution and processor workloads. However, these assumptions may not hold at query execution time. In this paper, we propose a dynamic query processing scheme and we present the performance results obtained by simulation of a queueing network model of the proposed software architecture.

Key Words & Phrases: database machines, dynamic query processing, performance analysis.
1985 Mathematics Subject Classification: 69C40, 69C24, 69H24, 69H26, 69H33
1990 CR Categories: C.4, C.2.4, H.2.4, H.2.6, H.3.3, I.6.3

1 Introduction

Exploitation of distributed processing capacity provided by large multi-processor systems remains one of the cornerstones to improve the performance of database management systems. The key to achieve this goal lies in the architecture of the query optimizer and the query execution strategy. To illustrate, several large research groups have produced research prototypes that attack the problem from a different angle. To name a few:

The distributed query processing technique of Bubba [ea90] is a multi-stage set-at-a-time technique. That is, the levels in the query plan lead to several stages at run time, where the transport of intermediate results between the layers in the plan is based on

*The work reported in this paper was conducted as part of the PRISMA project, a joint effort with Philips Research Eindhoven, partially supported by the Dutch "Stimuleringsprojectteam Informaticaonderzoek (SPIN)".

Report CS-R9050
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

declustering the tuples first. Moreover, the program components to solve the query are dynamically loaded upon need at each processing step.

The query processing techniques of PRISMA [KAH+88] and GAMMA [DGS+90] are based on a pipelined query processing technique. During query optimization the pipe layout is determined and its junctions are mapped to processing nodes. Subsequently, the query is solved in a dataflow driven manner.

Although each prototype has demonstrated performance improvement over centralized query processing, they are mostly based on the same hidden assumptions, which may still block a potential leap in performance for large scale multiprocessors.

The predominant assumptions are that a query optimizer generates the single optimal plan of action and that it does not take into account the possibly disastrous effect of concurrent running queries. For example, concurrent queries may have an exclusive lock on a relation fragment, which may cause a delay in the estimated response time. This knowledge could be used to switch to an alternative plan at runtime, which would exploit the waiting time to produce a partial result already in another way.

Furthermore, the query plan is based on simple estimates about the data distribution. A plan is normally not adjusted when these estimates turn out to be wrong. At best a query processor aborts the query when one of the subqueries produce an empty result or the query optimizer regularly refreshes its statistics. Ideally, the query processor could dynamically adjust the plan upon recognition of major deviations from the statistics maintained. An approach in this direction is also presented by [GW89].

The query plan generated in PRISMA (to a lesser extend in Bubba and Gamma) is based on an a priori known number of processors. For example, a query that joins three relations can be assigned 5 processors for pipelined processing. Equally, the query optimizer can design a plan that uses tens of processors using a declustering scheme on the relations and intermediate results. Unfortunately, either plan is fixed, which means that all resources should be conceptually acquired during query startup to guarantee the response time aimed for. In particular, the database should be properly declustered before, such that parallel execution for a large class of queries pays off.

Fixing the number of processors within the query plan also ignores the relative speed by which the operands produce their (partial) answers. For example, to deal with a bursty behavior of the processes involved, it would be better to automatically acquire more processors when work is available and release processors when nothing is left to be processed.

The dynamic query processing scheme described in the sequel is based on the observation that the strong relationship between query plan and query execution should be weakened. Instead, we propose to consider query processing a dual problem. First, how to solve a query when you have a (small) portion of the database within the query program buffers. The techniques for this can be borrowed from centralized query processing. Second, how to collect the interesting portions and how to distribute the work over the available processors using a centralized scheduler.

The core of this paper is an investigation in the potential performance bottlenecks that might arise from this scheme. We have constructed a queueing network model, to gain insight in the behavior of our query processing scheme. Using a parameter setting that reflects the properties of a reasonably tuned distributed operating system and main-memory DBMS implementation, we conclude that a central scheduler does not become an immediate bottleneck.

Our architecture also indicates an alternative caching strategy for database systems, based on dynamic replication of database fragments during query processing. Finally, the model indicates that reasonable linear speedup for processing join queries on a PRISMA-like database machine is attainable.

The remainder of this report is organized as follows. In Section 2, we describe a global architecture for our dynamic query optimization scheme. Section 3 describes the performance model and the simulation parameters. The basic queueing model is introduced in Section 4, while an improved model to deal with distributed caching is given in Section 5. We conclude with a summary and an indication of future research.

2 The Qstar architecture

In this section, we give a system architecture for our dynamic query processing scheme. Its main purpose is to show an evolutionary path for the PRISMA system architecture in sufficient detail. Furthermore, this architecture is used as a reference model for the performance analysis and a mock-up implementation to validate our approach. The system's nickname is Qstar, which indicates the central role of individual queries.

The Qstar design objectives diverge from the PRISMA approach in several aspects. First, we do not assume that there exists a query optimizer that produces uniformly good, yet static query plans. Rather, we aim for a dynamic query optimization strategy.

Second, we favor repetitive queries over ad-hoc, i.e. one shot queries. That is, we believe that most queries in practice are recurrent in nature. That is, successive calls only differ in the constants included. This warrants a compiled approach to gain performance.

Third, we foresee that most applications are browsing in nature, i.e. selecting a few tuples followed by a lengthy delay during which the user (application) absorbs the data. This calls for mechanisms that not only control the order and speed of presenting the tuples to the application program (i.e. cursor control), but also those that control the actual query resolution without blocking resources in the database (i.e. memory and processors) for lengthy periods.

The global system architecture is modelled after PRISMA and consists of three layers (Figure 1). The bottom layer is a collection of One Fragment Managers (OFM) and a concurrency controller. Each OFM is a relational data manager geared towards supporting a single horizontal relation fragment. In particular, it provides a relational algebra interface and a checkpoint/recovery facility. The OFMs do not contain any knowledge about other OFMs. Rather, their orchestration is handled at the distribution level.

The distribution level of Qstar differs from PRISMA in the following aspects. The query entered by the user is processed by the SQL parser, analyzed for semantics errors, logically optimized, and, finally, a code generator produces a Query Processing Monitor (QPM) and a Query Processor Object (QPO). A QPO contains the algorithms to solve the query under

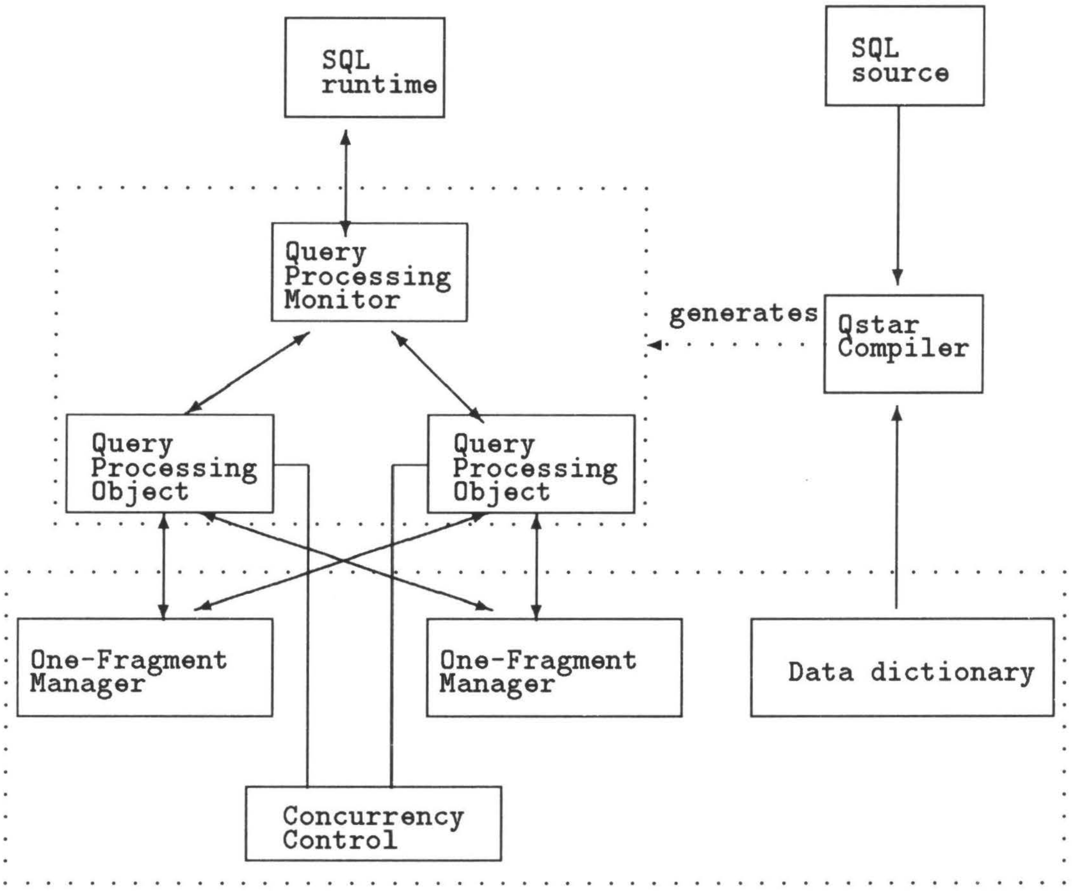

Figure 1: The Qstar architecture

the assumption that (all) data is available within the local address space of the process, i.e. a main-memory buffer pool. Moreover, it contains an interface with the One Fragment Managers to extract (replace) portions from (in) the database. Each QPO can thus be seen as a query specific server. Several may be active in handling a single request, while any number of QPOs may have been installed on a processor pool.

The QPOs are managed by a single Query Processing Monitor, which contains a strategy to obtain database portions from the OFMs and which distributes the workload over the QPOs. The QPM, comparable with a transaction manager, is also given the responsibility for distributed integrity- and concurrency- control. A more detailed description of the OFM, QPO, and QPM is given below.

2.1 One-Fragment manager

The base relations are managed by a so-called One Fragment Managers (OFM). Like in PRISMA they are always optimized towards supporting a single relation fragment. That is, an OFM is ideally compiled from a single (extended) SQL create statement.

Each OFM is implemented as a server object, that contains at least two kinds of threads. A communication thread and a storage thread. The communication thread handles the communication with the clients, such as queues incoming requests for execution and enquiries. The storage thread deals with storage subsystems, such as the file servers, and it evaluates the relational subqueries.

To illustrate, consider the case that a QPM requests a range selection on the base table. That is, the OFM receives the message *select segment*, which is queued upon arrival. Once a storage thread is scheduled for execution, it picks a request from the queue of pending queries and prepares a local processing plan. This plan is subsequently taken into execution and the qualifying tuples are copied into a tuple segment. Once the tuple segment is full, the client is notified with the message *segment cached* .

The tuple segment itself is not being sent, but it remains cached in the OFM. It can be obtained by interested parties later on by issuing a *get segment* message, which is handled directly by the communication thread. The tuple segment is set up such that no additional copying is needed to reply.

Associated with each tuple segment we keep reconstruction information, such as the range query, the base table portion being used, and the cost involved in re-producing it. When a tuple segment should make room for a new one, this information can be used to decide whether to copy the partial result to disk or to reconstruct it when need arises.

2.2 The Query Processing Object

The prime task of a Query Processing Object is to evaluate a query upon receiving the message *reduce vector* from the Query Processing Monitor. A segment vector consists of a list of tuple segment id's, one for each database variable mentioned in the query. Upon receipt, the QPO first acquires copies of the tuple segments mentioned in the message. This

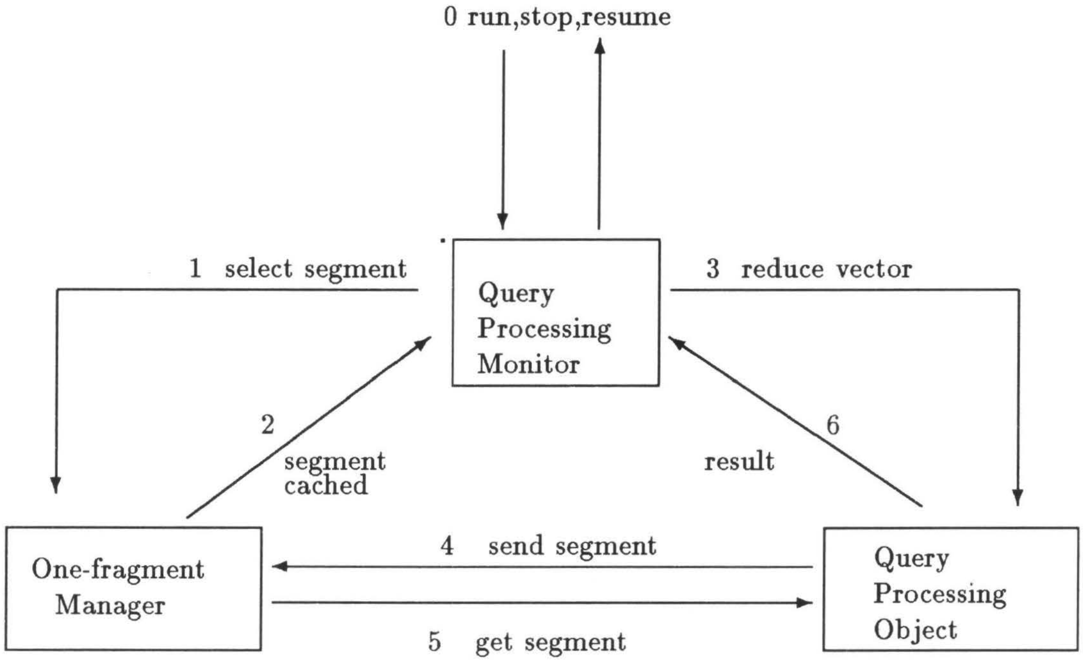

Figure 2: The Qstar communication pattern

involves communication with one or more OFMs when the tuple segment was not already cached in the workspace of the QPO itself.

The QPO algorithm should be designed such that partial results are produced as quickly as possible with minimal space consumption. For example, a hash-index can be associated with each incoming tuple segment. When all segments have arrived the join algorithm can use them to quickly discard non-qualifying tuple combinations.

Following, two strategies can be applied to produce a result. First, the traditional approach is to actually construct an intermediate relation, which contains only the target tuples. This involves joining the tuple segments, and copying the qualifying tuples into tuple segments. This intermediate is presented to the outside world analogous to the base tables.

Second, result construction is delayed until it is actually needed in the application. Therefore, for each database variable a new segment is produced that contains the tuples that participate somehow in the result. The final phase, i.e. construction of the target tuples, is done within the application workarea.

The prime advantage is that the communication overhead for emitting the result is never greater than the sum of its input. The drawback is that the actual relationship among the tuples should be reconstructed by building indices again. We expect that re-construction is less expensive than shipping indices.

2.3 The Query Processing Monitor

The prime task of the Query Processing Monitor (QPM) is to distribute the workload over the OFMs and the available QPOs to produce the partial results. The communication pattern is shown in Figure 2. The *run*, *stop*, *resume*, and *abort* are user commands. The message *select segment* tells the OFM to produce a new tuple segment. The OFM replies with the message *segment cached*, when it has read a segment from disk. The message *reduce vector* tells a QPO to produce a reduced copy of the operands, such that no tuples are retained that provably will not be part of the final result. When the QPO has completed this task it will respond with the message *result.*

The QPM itself is divided into three major components; the pairing-, the filter-, and the schedule - component. They will be discussed in following paragraphs using the query to join two Wisconsin 10K relations, relA10k $\bowtie$ relB10k.

Assume that the OFMs involved in maintaining the base relations have received a request to produce a tuple segment. Then, in due time, they will notify the QPM that such a segment has been *cached.* Say, the first segment of relA10k is cached, denoted by relA10k[1], which is recorded by the QPM.

When the message *segment cached* arrives from the other OFM, say relB10k[1], then all possible pairings are made with previously cached relA10k segments. In this case, a task *reduce vector* is formulated to reduce the tuple segments by evaluating the query for relA10k[1] and relB10k[1]. This task is stored in a work table. Once the result becomes available its status is turned into *cached.* This pairing algorithm is applied to all segments being cached. The effect is formation of the Cartesian product over all relations operands in the query, i.e. a nested-loop over tuple segments. To illustrate, the table below contains a portion of the administration kept by the QPM.

cached	relA10k[1]
cached	relB10k[1]
select	relA10k[2]
cached	relB10k[2]
select	relB10k[3]
reduce	relA10k[1] relB10k[2]
cached	relA10k[1] relB10k[1]
...	

To improve the response time of a query considerably we could filter the tasks to avoid firing useless (sub)tasks. This filter uses semantic knowledge to reduce the Cartesian search space generated by the pairing component. For example, we might use min/max values over the join attributes supplied by the OFM to drop vectors that do not overlap. The net effect is to take only those tasks into execution that contribute to the query result.

The filtered vectors are handed to the scheduler that maintains a table of available QPOs. For each QPO it knows the segments already stored there and the amount of slack resources. Thus, it can direct the vector to the least costly QPO, i.e. cheapest in communication.

3 Performance Analysis Method

The performance analysis of the Qstar architecture is based on modelling it as a queueing network. In particular, for each component in the system architecture we could identify an arrival time distribution function (A), a service time distribution function(B), the number of services(n) and the service discipline(d). This leads to an element of the queueing model space (A/B/c/d), using Kendall's notation [Kle75].

Unfortunately, queueing theory does not yet provide answers to all possible combinations, nor is it possible to cast the behavior of Qstar in a set of closed formula. Therefore, we have used the queueing network model as a basis for a discrete event simulation, where the characteristics are partly described by distribution functions and partly described by algorithms. Following, the simulation scheduler can collect the relevant measurements of the interesting performance indicators.

The queueing network modelling package QNAP2[1]version 5.0 has been used [CII84]. This package contains algorithms for discrete event simulation and algorithms for analytic solutions. The analytic solvers yield exact and approximate steady-state solutions provided the simulation model satisfies some severe constraints, such as, mutual independence between stations.

The major limitation of a simulation is its production of average indicators instead of exact or approximation results and the processing time involved to gain stable results, which limits the number of cases that can be covered. (Some of the runs lasted for days.) Furthermore, the complexity of the QPM is reduced by ignoring the filtering component. The result on our simulation is that the performance figures indicate a 'worst-case' situation, because there is no semantic feedback in the system to reduce the amount of work generated within the QPM.

3.1 Performance measurements

The focal point of our performance analysis are the questions *what is the Qstar system utilization* and *will Qstar exhibit a linear speed up for large number of processors.* To answer these questions we acquired several performance factors from our simulation.

The prime simulation variable is the number of QPO centers, because it relates to the fraction of the PRISMA machine that could have been used. Furthermore, we are interested in the following output measures:

- system measures
  - system throughput (in terms of processed segment vectors per second)
    This will be an indication for query response time. The number of vectors estimated during query compilation divided by the system's throughput yields the response time.

[1] QNAP2 (Queueing Network Analysis Package 2) is copyrighted by CII HONEYWELL BULL and INRIA 1981,1982

  - average number of messages in system
    This will be an indication for the network load. Too many messages cause network congestion.

- service center measures (QPM, OFM, QPO)
  - utilization of a center
    This will be an indication for a possible bottleneck in the system. A relative high utilization percentage, that increases with each additional QPO, will limit the performance in the long run.
  - average residence time
    This represents the waiting time in a queue and in the service center of a station.
  - throughput
    This highlights centers for further improvement
  - average queue length
    A high performance implementation will be infeasible, if this is accompanied by large queues at stations and show insufficient buffering capacity.

We limited the amount of data gathered from the simulation to the messages that requested joined segment vectors.

3.2 The parameter settings

The parameter settings for our simulation are not directly derived from the PRISMA machine, because no stable figures were available at the time of writing. Instead, we have used parameter settings that reflect a reasonably optimized distributed operating system and we used our experience in writing a main-memory OFM early on in the project. As such, the parameter settings reflect a system that could emerge within the next few years. A more detailed validation of the parameter settings for the queueing model is underway.

The test query we will be working with is as follows:

$$\mathbf{relA10k} \bowtie \mathbf{relB10k} \bowtie \mathbf{relC10k}$$

We begin with the mean time to transport data over the network. These values are based on Amoeba network characteristics [MvRT+90].

The setup costs for communication	1.0	ms
Transfer costs of data (per Kb)	1.1*S	ms
Total costs for communication	1 + 1.1 * S	ms

The parameter 'S' represents the data size in Kb. Since communication is negatively influenced by small blocks, we use a segment size of 32 Kb. Although the Qstar query processing scheme could request the OFMs to project out the unwanted attributes from the relations, we assume in our simulation that this has not been done. Consequently, each segment contains about 160 tuples.

We further assume rather lengthy RPC messages of 250 bytes. These are needed to send the segment vectors around. The corresponding delays introduced by the network are:

for a segment transfer	32*1.1+1	=	36.2	ms
for a RPC	1 + 0.25*1.1	=	1.3	ms

The mean time for an OFM to prepare a segment, i.e. local query optimization and index lookup, is related to the segment size and pre-selection cost within the OFM. For the latter aspects we used a fixed cost of 20 ms. During selection the qualifying tuples are be copied into a result tuple segment. The processing time is calculated as follows:

Time to prepare a tuple segment	20.0	ms
Time to move data around in memory (per Kb)	0.2 * 32	ms
Total costs for preparing a segment	26.4	ms

The task of a QPO is to handle a M-way join. The costs involved are based on a hash-join algorithm that consumes 0.1 ms per tuple. We assume that each QPO produces a single segment that contains 5% of the input tuples. The processing time of a QPO over M relations becomes:

M * hash-index construction	M * 160 * 0.1	=	16.0 * M	ms
one join scan to collect result		=	0.1 * M	ms
5% tuple construction	M * 0.2 * 0.05 * 160 * 0.200	=	0.32 * M	ms
Processing time to join M relations			16.42 * M	ms

The delay from the network for transferring the result becomes:

Transfer of result	5% from 160 tuples			
Size of result	8 * 200 * M bytes			
	1.6 * 1.1*M + 1	=	1.8*M + 1	ms

The experiments are conducted with 3 OFMs (3-way relation join), thus:

Time to join 3 relation segments	3*16.42 ms	=	49.26	ms
Time to transfer result over the network	1.8*3 + 1 ms	=	6.4	ms

The PRISMA machine has a fully interconnected network architecture, where each link has a bandwidth of 20 Mbits/sec. The simulations are performed under these network assumptions.

4 Qstar queueing model

In this section we describe the general queueing network layout of Qstar, the message classes and the modelling assumptions. In the remaining sections we illustrate an analytic model for the components involved. The outcome of this section is a basic model that captures the message flows and the processing characteristics of the basic Qstar implementation. Some illustrative performance figures are given to highlight its behavior under the parameter settings. In the next section we will improve the basic model by using the caching of segments within QPOs to enable segment exchange between QPOs.

4.1 Message classes

The components of the Qstar architecture are mapped to service centers in the queueing network. That is, we have a single service center to model the QPM, a fixed number (m) of OFM service centers that hold the query operands, and a variable number (n) of service centers for the QPOs. In addition, we assume a service center that mimicks the user input, called SRC, and a service center for the communication network, that models the network delays. The corresponding message classes and propagation paths are as follows:

- *class-0* $SRC \rightarrow QPM$
  This class marks the beginning of a new vector evaluation.

- *class-1* $QPM \rightarrow OFM_i \rightarrow QPM$
  The QPM produces m *class-1* messages for each *class-0* messages to request a selection by the i-th OFM.

- *class-2* $QPM \rightarrow QPO_j \rightarrow QPM$
  The QPM converts the answers by the OFM into segment vector evaluation requests to be sent to the QPOs. Upon receipt of the answer it marks the vector as being cached.

- *class-3* $QPO_j \rightarrow OFM_i \rightarrow QPO_j$
  In a QPO the vector request is turned into a request to the OFMs to deliver the necessary segments for processing.

To avoid unnecessary complications we have not included the network component in this list. Actually all paths go through the network service center. The exception is the $SRC \rightarrow QPM$ link which is internal to the QPM.

Furthermore, the SRC includes a heuristic to avoid over- and underflowing the QPM center with (user) requests. That is, the source will keep on supplying requests until the number of segments already at the QPM center exceeds twice the number of idle QPOs. Thereafter, it reduces to a slow rate to avoid overflowing QPM.

4.2 Model assumptions

The analytic models derived for the system components are based on the assumption that service times for all classes within a service center are exponentially distributed and the service discipline is First Come First Serve. Furthermore, the communication is asynchronous, which avoids blocking the individual service centers (deadlock as well). Finally, we assume infinite buffering capacity and sufficient bandwidth to handle the requests. The latter assumptions are quantified in the subsequent simulations, which show that indeed this assumption holds for PRISMA-like machines.

4.3 Analytical model

The Qstar components can be individually characterized with an analytic model. For brevity we will illustrate the modeling of QPM only. The models for OFM QPO, and network have been obtained in a similar fashion.

The QPM center deals with three message classes. The *class-2* messages communicate with the QPO and the *class-1* messages with the OFM. Furthermore, it receives the *class-0* messages produced by an internal source to mark the start of a new vector evaluation.

We assume that the system consists of independent Poisson processes, thus the total input process will be a Poisson process as well [Kle75]. Therefore, the overall interarrival time distribution of input messages is exponential with factor λ_{qpm}. We further assume that the service time distributions for the message classes is exponential, that is

$$s_i(x) = \mu_{qpm,i}\ e^{-\mu_{qpm,i}x}.$$

The parameter $\mu_{qpm,i}$ models the following aspects:

- $\frac{1}{\mu_{qpm,0}}$: the mean time to prepare a request for a vector for the OFM.
- $\frac{1}{\mu_{qpm,1}}$: the mean time to prepare an evaluation request for the QPO.
- $\frac{1}{\mu_{qpm,2}}$: the mean time to register a join of the segments within a vector.

The choice of our distribution functions makes the service time of the QPM center hyperexponential (H_3) [Kle75] namely:

$$b(x) = \sum_{i=0}^{2} \alpha_{qpm,i}\ \mu_{qpm,i}\ e^{-\mu_{qpm,i}x}$$

The distinct $\alpha_{qpm,i}$'s denote the fractions of the messages of the corresponding calls-i in the input stream. The average interproduction time is:

$$\overline{x}_{qpm} = \sum_{i=0}^{2} \frac{\alpha_{qpm,i}}{\mu_{qpm,i}}$$

The resulting output rate will be:

$$\mu_{qpm} = \frac{1}{\overline{x}_{qpm}}$$

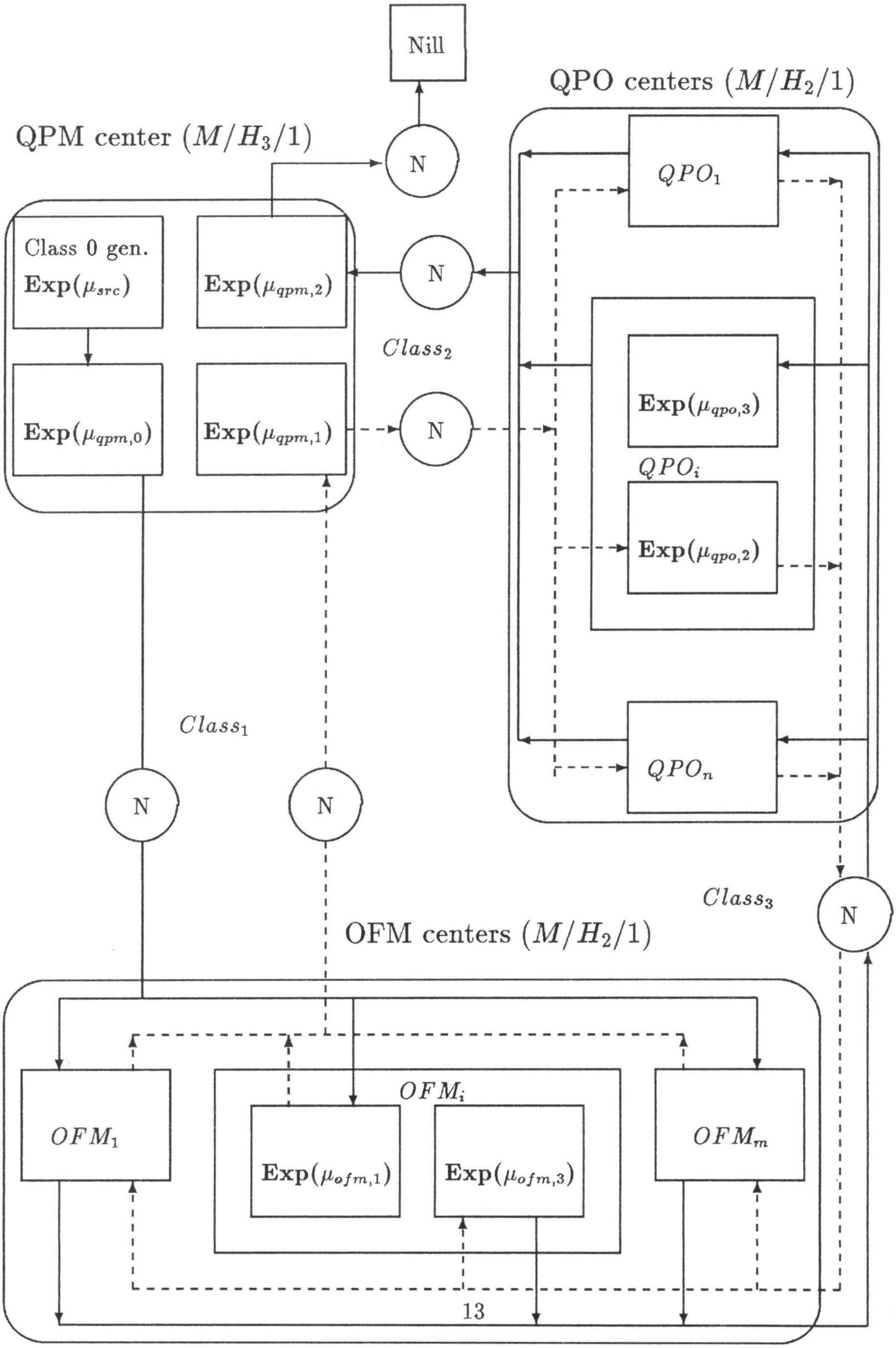

Figure 3: Basic model

To summarize the specifications for the QPM center (with $i = 0, 1, 2$):

- total input rate factor: λ_{qpm}
- the arrival time distribution function : $\exp(\lambda_{qpm})$
- probability message of class i in the input stream : $\alpha_{qpm,i}$
- service time distribution class i message : $\exp(\mu_{qpm,i})$
- total service distribution: *hyperexponential* (H_3)
- average output rate : μ_{qpm}
- per class the messages are directed to an unique destination
- we classify the QPM center as : $(M/H_3/1)$
- service discipline: FCFS

The analytical models for the OFM and QPO are developed along the same line. The input streams are again considered independent Poisson distributions leading to hyperexponential distributions as well. An overview of the resulting queueing model is shown in figure 3.

4.4 The network service

The network service process represents the transfer delay encountered during interprocess communication. A routing mechanism for messages has been implemented in the simulator. The mechanism is needed for modelling the communication, such as to impose a delay upon message transport. For example, a *class-2* message send from the QPM to a QPO process must encounter a different delay than the returning message, which contains data and is larger.

All processes at a center will have the same probability of being chosen as a destination process. This simplifies the routing mechanism by only distinguishing centers. A process in the destination center is chosen randomly by the network to become the receiving process. Without this assumption every path between processes should have been made distinguishable by an unique class, because routing can not be specified in such details in an analytical queueing network model. The processes involved are again modelled as Poisson processes, leading to hyperexponential distributions.

5 Evaluation

In the following we present the results obtained by running a simulation of the basic model. The results of this experiment (see Section 5.1) show that the load of the different processes is not equally distributed. The basic model is improved by introducing a segment exchange

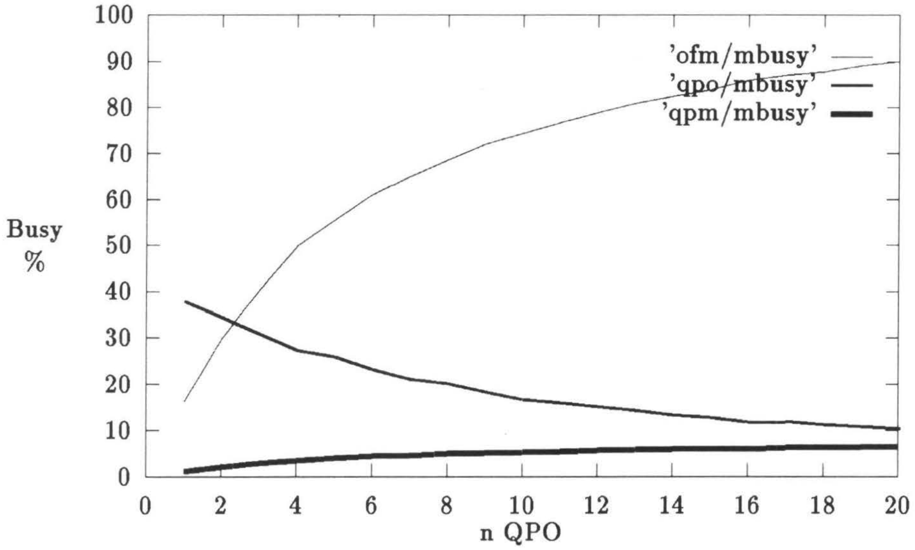

Figure 4: Utilization level

mechanism between QPOs. This extended model shows a better load distribution. The results can be found in Section 5.2.

5.1 Basic model

The simulation results are shown in Figure 4 and 5. The simulated time has been set high, so as to obtain statistical measures with a 95% confidence intervals of ± 10% around the mean.

Figure 4 shows the utilization levels for the three system components. In this architecture, the load on the OFMs is much higher than on the QPM and QPO. In particular, its reaches 90% utilization with 20 processors already. The underlying cause is that the ergodic constraint at the OFMs reaches equality at this point ($\frac{\lambda}{\mu} \approx 1$). That is, the expected number of arrivals (λ) at a center reaches the serving capacity (μ).

An indication of the total throughput for each process is given in Figure 5. The central role of the QPM is highlighted by the throughput of messages through this center. Furthermore, the bottleneck is the OFM which can not adequately handle an increasing number of QPOs. Therefore the throughput for QPOs decreases.

5.1.1 Network measures

The assumption that the network will not limit the system still holds, as can be seen in table 2. Using the message type distribution, obtained from the simulation, and the size of each message type (Table 1), we have calculated in this table the total data throughput for each process. It turns out that for all process types the required bandwidth is much lower than the network limit of 2 Mbytes/sec. Thus the capacity of a single network link is not

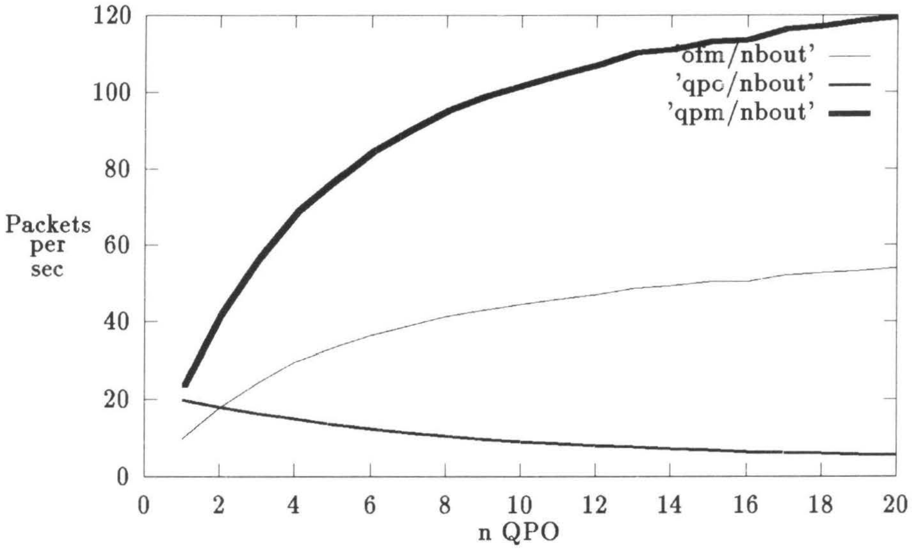

Figure 5: Throughput of messages

Message type	size
RPC	0.5 Kb
Data	4.8 Kb
Segment	32.0 Kb

Table 1: Messages sizes

exceeded. Under the assumption that a fully interconnected network is used, we conclude that the global network bandwidth is not exceeded either.

5.2 Extended model

In the previous section we observed that the OFM forms the potential bottleneck in the system. The congestion of the OFMs can be avoided by also using the QPOs as a cache for the tuple segments, thereby spreading the load for accessing tuple segments over both the QPOs and OFMs. To simplify the model, we assume unbounded caching resources at the QPOs. In the remainder of this section we show the results of a simulation of this model.

This model was simulated under the same conditions as the previous model (i.e. accuracy, simulation time). The results from these runs are presented in Figure 6 - 8.

In Figure 6 the utilization level of the three system components is shown. The bottleneck has been shifted towards the QPM, which reaches saturation with 60 QPOs. However, the throughput peak of the QPM is reached for a much lower number of QPOs (43), which means that between 43 and 60 one already faces a reduced payoff of parallel execution.

Compared with the simple model we have doubled the effective number of active QPOs and we obtained a 5 times higher throughput (20 QPOs) by better utilization (See figure

Process	I/O	Description	Type	Messages/sec	Kb/sec
QPM	IN	*segment cached* from OFMs	RPC	82.5	
		result from QPOs	Data	27.0	
					170.9
	OUT	*reduce vector* to QPOs	RPC	27.0	
		select segment to OFMs	RPC	82.5	
		partial result to user	Data	27.0	
					184.4
					355.3
QPO	IN	*reduce vector* from QPM	RPC	0.7	
		get segment from OFMs	Segment	2.1	
					67.6
	OUT	*result* to QPM	Data	0.7	
		send segment to OFMs	RPC	2.1	
					4.4
					72.0
OFM	IN	*select segment* from QPM	RPC	27.5	
		send segment from QPOs	RPC	27.0	
					27.3
	OUT	*segment cached* to QPM	RPC	27.5	
		get segment to QPOs	Segment	27.0	
					877.8
					905.1

Table 2: Network requirements for the basic model

8). Furthermore, the model displays linear speedup in processing up to 30 QPOs.

5.2.1 Network measures

As with the simulation of the basic model we verify the network assumption using the simulation results. The calculations of the required network bandwidth for each processor link can be found in Table 3. We see that although the maximum vector throughput has increased considerably (Figure 8), that the network throughput for each process does not exceed the maximum link bandwidth.

5.3 Response time

The simulation results can be used to obtain an indication of the response time for the example query. Recall that it represents a three-way join over the Wisconsin relations without support of access paths and without reduction of the operands by pre-selection. Each relation contains 10K tuples, which leads to about 60 segments per operand. Thus, a naive implementation of the QPM would produce 216000 segment vectors (60^3).

This job can be handled with 20 parallel QPOs without overloading any of the processors. Their utilization level is about 30-40%. For this system configuration we need about

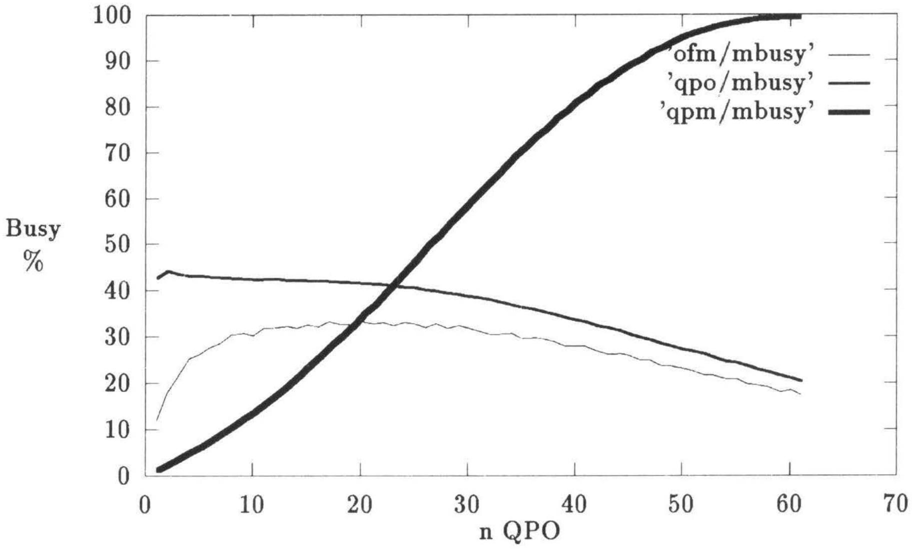

Figure 6: Utilization level

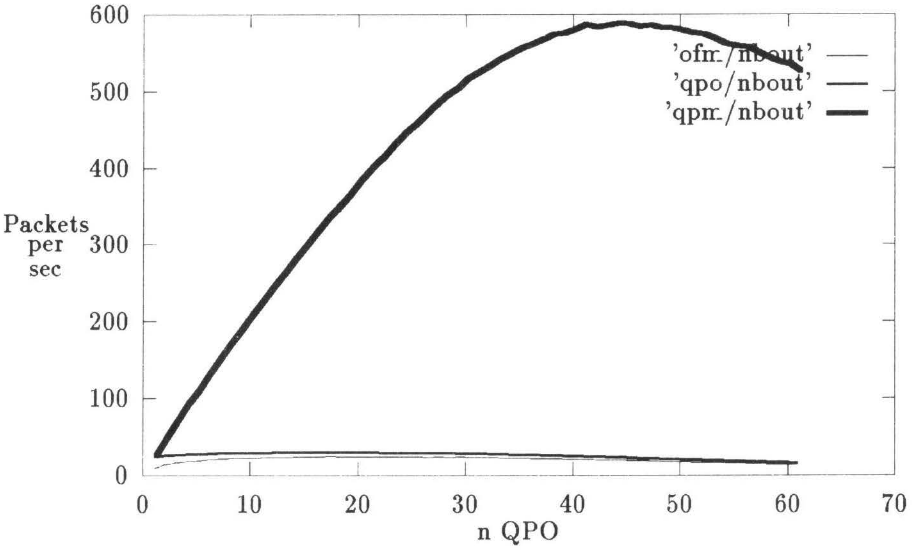

Figure 7: Throughput of messages

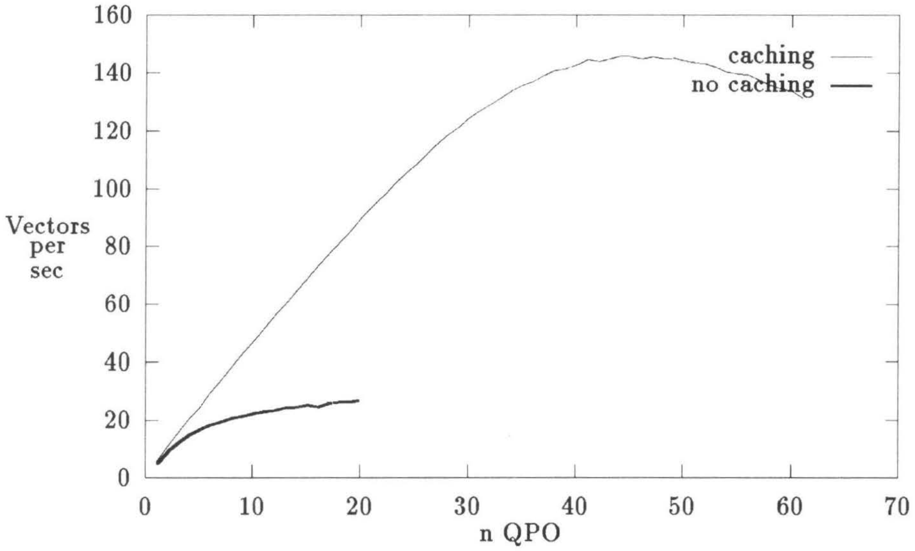

Figure 8: Total number of joined vectors for both models.

11ms per vector (Figure 8), which leads to a response time of 2376 seconds. A response time of 1512 seconds can be obtained, if 40 QPOs are used. Clearly, this is not competitive with current commercial systems, because it is essentially based on the nested-loop join algorithm.

Yet, the response time can be improved dramatically when we spent little more time in preparing the work within the OFM using well-known techniques. For example, the OFM may be requested to hash-partition the relation into batches of 10 segments. This leads to an initial delay of 0.26 second before the first segment batch becomes available. The advantage is that now the QPM filtering component can drop all vectors that have incompatible hash values. After the first batch of each OFM we can form 10 tasks, activating 10 QPOs as well. The second burst of each OFM will expand the vector table to 10.2^3, generating enough work to keep 40 QPOs active. Afte: the k-th burst it will be $10.k^3$. In our example, we can hash partition the contents of the OFM of 60 segments into k= 6 batches. This results in a total amount of 2160 vectors with an expected response time of about 15 seconds. (The work within the OFM after $k = 2$ is overlapped by the processing in the QPOs.)

6 Summary

In this paper, we have presented an alternative approach for query processing on large multiprocessors. Our approach is based on breaking the query into two smaller problems, namely, how to solve the query for a small portion of the database and, how to schedule a large number of tasks, which together form the query program. The hypothesis is that the combination leads to better system utilization and smooths the fluctuations normally encountered in parallel query processing.

Process	I/O	Description	Type	Messages/sec	Kb/sec
QPM	IN	*segment cached* from OFMs	RPC	26.9	
		result from QPOs	Data	75.2	
					374.4
	OUT	*reduce vector* to QPOs	RPC	75.2	
		select segment to OFMs	RPC	26.9	
		partial result to user	Data	75.2	
					412.0
					786.4
QPO	IN	*reduce vector* from QPM	RPC	1.6	
		get segment from OFMs and QPOs	Segment	4.8	
		send segment from QPOs	RPC	10.5	
					159.7
	OUT	*result* to QPM	Data	1.6	
		send segment to OFMs and QPOs	RPC	4.8	
		get segment to QPOs	Segment	10.5	
					346.1
					505.8
OFM	IN	*select segment* from QPM	RPC	9.0	
		send segment from QPOs	RPC	9.1	
					9.1
	OUT	*segment cached* to QPM	RPC	9.0	
		get segment to QPOs	Segment	9.1	
					295.7
					304.8

Table 3: Network requirements for the extended model

A queuing network model has been constructed that captures the processing aspects of our architecture. It has been used to drive a discrete event simulation to experiment in a time efficient way with two processing strategies, i.e. a central and decentralized caching of tuple segments.

The two simulations show that the central scheduler does not lead to an immediate bottleneck. The linear speed-up curve flattens before the scheduler becomes overloaded. That is, the speedup from parallelism becomes neglectable before the QPM becomes saturated. Furthermore, the decentralized caching of segments proved effective.

The system utilization in both cases is still limited, mainly due to the network activities, which are modelled as independent processes. Thus, once a QPO has issued a request for a tuple segment it has to wait for delivery; it does not take part in handling the communication protocols.

Designing the filter algorithm as well as query specific scheduling is an open-ended track. The filter can do a better job once more feedback information is passed to the QPM about the contents of the segments being cached. For example, as part of the message *cached* one could also return the min/max over the join attributes. This would enable the filter to precompute the overlap of a proposed segment pairing (and drop it when no such overlap exists). Furthermore, one can easily configure a more static evaluation plan within the QPM to enforce a specific order of vector evaluation.

A lot more has to be done. The current activities are focussed on a validation of our simulation in the 'real' multiprocessor environment. Furthermore, a comparison with architectures based on static query plans is under way.

References

[CII84] CII Honeywell Bull and INRIA. *QNAP2*, 1984. Introduction to QNAP2 and Reference Manual.

[DGS+90] D. J. DeWitt, S. Ghadeharizadeh, D.A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Transactions On Knowledge and Data Engineering*, 2(1), March 1990.

[ea90] H. Boral et al. Prototyping bubba, a highly parallel database system. *IEEE Transactions On Knowledge and Data Engineering*, 2(1), March 1990.

[GW89] Goetz Graefe and Karen Ward. Dynamic query evaluation plans. In *Proc. SIGMOD*, 1989.

[KAH+88] M.L. Kersten, P.M.G. Apers, M.A.W. Houtsma, E.J.A. van Kuyk, and R.L.W. van de Weg. A distributed, main-memory database machine; research issues and a preliminary architecture. In M. Kitsuregawa, editor, *Database Machines and knowledge Base Machines*, pages 353–369, 1988.

[Kle75] Leonard Kleinrock. *Queueing Systems, Theory*, volume 1. John Wiley & Sons, 1975.

[MvRT+90] S.J. Mullender, G. van Rossum, A.S. Tanenbaum, R. Renesse, and H. van Staveren. Amoeba - a distributed operating system for the 1990s. *IEEE Computer Magazine*, May 1990.