# Affinity Scheduling in Staged Server Architectures

Stavros Harizopoulos and Anastassia Ailamaki

March 2002
CMU-CS-02-113

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

**Abstract**

Modern servers typically process request streams by assigning a worker thread to a request, and rely on a round robin policy for context-switching. Although this programming paradigm is intuitive, it is oblivious to the execution state and ignores each software module's affinity to the processor caches. As a result, resumed threads of execution suffer additional delays due to conflict and compulsory misses while populating the caches with their evicted working sets. Alternatively, the staged programming paradigm divides computation into stages and allows for stage-based (rather than request thread-based) cohort scheduling that improves module affinity.

This technical report introduces (a) four novel cohort scheduling techniques for staged software servers that follow a "production-line" model of operation, and (b) a mathematical framework to methodically quantify the performance trade-offs when using these techniques. Our Markov chain analysis of one of the scheduling techniques matches the simulation results. Using our model on a staged database server, we found that the proposed policies exploit data and instruction locality for a wide range of workload parameter values and outperform traditional techniques such as FCFS and processor-sharing. Consequently, our results justify the restructuring of a wide class of software servers to incorporate the staged programming paradigm.

Email: {stavros, natassa}@cs.cmu.edu

Stavros Harizopoulos is partially sponsored by a Lilian Voudouri Foundation Fellowship and gratefully acknowledges their support.

**Keywords:** DBMS, databases, performance, staged server, scheduling, cache-conscious.

# 1 Introduction

Modern application servers typically accept streams of requests, and assign each request an execution thread. Context-switching amongst threads is at the discretion of the operating system, which typically employ a time-sharing context-switching policy. This programming paradigm is elegant and intuitive; however, each request spends a different amount of time in each software module. Round-robin time-sharing context switching is oblivious to the execution state, and ignores module affinity to the memory hierarchy resources. Consequently, each resumed thread often suffers additional delays while re-populating the processor caches with its evicted working set.

As the processor-memory speed gap [12] continues to increase, cache misses are becoming an increasingly important factor of a server's performance. Research [15] has shown that this gap affects commercial database server performance more significantly than it affects other engineering, scientific, or desktop applications. The reason is that database applications access the memory subsystem far more often than desktop or engineering workloads. Moreover, database workloads exhibit large instruction footprints and tight data dependencies that reduce instruction-level parallelism opportunity and incur data and instruction transfer delays [7][10].

To alleviate memory delays, numerous data placement techniques and cache-conscious algorithms [2][3][5][6] have been proposed that exploit spatial and temporal locality. Such techniques, however, improve the locality *within* each request and have limited effects on the locality *across* requests. Database servers typically assign threads to queries; therefore, context-switching across concurrent requests is likely to destroy data and instruction locality in the caches. When running OLTP workloads, for instance, most misses occur due to conflicts between threads whose working sets replace each other in the cache [8][9]. As incoming requests go through different modules of the server code, while the CPU switches execution between concurrent requests. Figure 1 illustrates an example where successive executions of module 2 are interleaved with executions of other modules, forcing module 2's content to be repeatedly evicted from and restored into the memory hierarchy. As future systems are expected to have deeper memory hierarchies, a more adaptive programming solution becomes necessary to best utilize the available memory resources.

The staged server [1] is a programming paradigm that divides the computation that a server performs in response to a request into stages and schedules request execution within one stage at a time, to improve locality. The staged server defers executing a stage until its queue accumulates operands, then it processes the entire queue, by repeatedly executing its operation. Once that queue is empty, the processor keeps traversing the list of stages going first forward and then backward. The authors have demonstrated that their approach can improve the performance of a simple, custom built web server by reducing the frequency of cache misses in both the application and operating system code. While the web server is a successful first

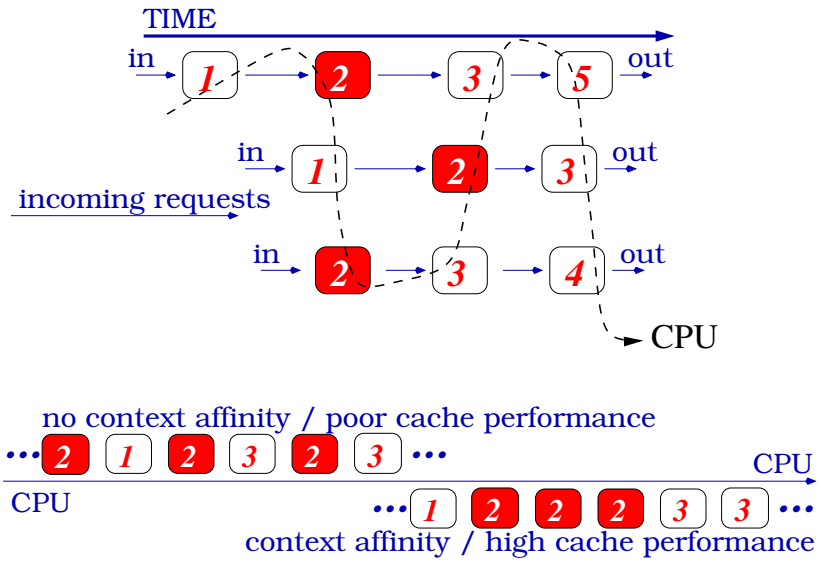


**FIGURE 1:** *Uncontrolled context-switching can lead to poor cache performance.*

experiment, there remain significant scheduling trade-offs to be solved. For instance, it is not clear under which circumstances a policy should delay execution of individual requests before moving to the next module, before the locality benefit is offset. In addition to the forward-backward list traversal scheduling algorithm proposed, other techniques may work even better under a variety of application workloads. The staged server paradigm can improve performance of an application server if (a) the application is bound by delays in the memory hierarchy, (b) requests are executed through well-defined modules, and (c) the module topology is relatively simple to minimize scheduling complexity.

Database management systems meet all of these requirements and are an excellent candidate for the staged programming paradigm. The discussion in the beginning of this section indicates that database applications suffer from memory hierarchy delays more than other types of workloads, which increases the potential for performance gains. Furthermore, database servers are naturally modular, as queries follow the same sequence (parser, optimizer, transaction manager, etc.). Different requests can benefit from locality within each module as they access common data structures and code. In addition, the flow of execution is purely sequential, which makes the search space for scheduling policies conveniently manageable: possible policies are more likely to include a combination of the number of requests to receive service (one, several, all), the time they receive service within a module (until completion or up to a cutoff value), and the order of visiting the modules.

To our knowledege, this is the first attempt to extensively study the scheduling trade-offs when applying the staged server paradigm in the context of a complex application server such as a database system. The contributions of this technical report are twofold. First, it introduces four novel cohort scheduling techniques for staged software servers that follow a “production-line” model of operation. Next, it presents

a mathematical framework to methodically quantify the performance trade-offs when using these techniques. We model one of the proposed policies using a Markov chain and validate its solution against simulation scripts. This analysis applies to a wide class of systems following a "production line" model of operation that may benefit from a stage divided service, repeatedly performed on batches of jobs. In the context of a simulated database system, we compare six scheduling alternatives: the traditional processor-sharing model, plain FCFS, and four proposed scheduling techniques. In order to show the performance gains under each possible scenario, we vary parameters such as the importance of the module loading time when compared to the execution time and the overall load in the system. Our results show that the proposed policies exploit data and instruction locality for a wide range of workload parameter values and outperform traditional techniques such as FCFS and processor-sharing.

The rest of the report is organized as follows. Section 2 reviews additional related work. Section 3 presents the framework along with our assumptions, defines the problem, and describes the existing and proposed scheduling policies. Section 4 presents a queueing model which incorporates locality-aware execution, along with its analysis. Section 5 discusses the results from measuring and comparing the performance of all policies under various scenarios. Section 6 discusses the system assumptions the simulation is based upon. Finally, Section 7 concludes the report's results.

## 2 Related work

The term "affinity scheduling" has been widely used in shared-memory multiprocessor systems. In those environments, it is sometimes beneficial to schedule a task on a certain processor that contains relevant data in its local cache [16][17]. Although this type of affinity is similar to the one that the staged programming paradigm tries to leverage on, the latter approaches the problem by restructuring a single application to exploit locality, rather than improve locality for a collection of tasks in a generic setting.

Most server architectures have adopted processor-sharing (PS) scheduling as a "fair" scheduling policy. The CPU(s) spend a fixed amount of time (typically in the order of 1ms) on each active process and keep switching processes in a round-robin fashion. Recent work tries to argue in favor of SRPT (Shortest Remaining Processing Time first) scheduling for the case of a single server [13]. The authors in [11] have shown that traditional assumptions for workload modeling are not accurate. More specifically, it was shown that UNIX jobs have sizes that follow a Pareto (heavy-tailed) distribution and not an exponential one. The effects of taking into account this distribution can be counter-intuitive and are discussed in [11].

In the database community there is significant research towards improving DBMS performance by making several of the database systems components "cache-conscious". Relevant studies target a part of the query execution lifetime such as query processing algorithms [5], index manipulation [2][3][4], and

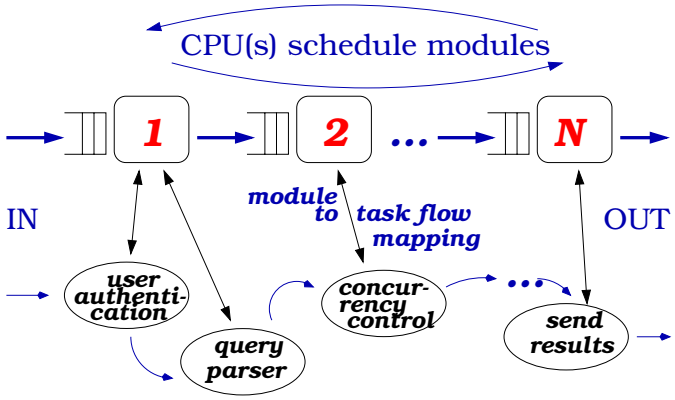


**FIGURE 2:** *A query passing sequentially through N modules.*

data placement schemes [6]. Although the proposed techniques take advantage of the inherent data and instruction locality within the database server modules to decrease cache misses (and hence memory delays), uncontrolled context-switching between different execution threads hurts program locality.

The authors in [14] proposed a staged event-driven architecture (SEDA) for highly concurrent internet services. SEDA decomposes a complex, event-driven application into a set of stages connected by queues. This design avoids the high overhead associated with thread-based concurrency models and enables services to be well-conditioned to load, preventing resources from being overcommitted when demand exceeds service capacity. The potential benefits of this design are orthogonal to the benefits stemming from a cache-aware scheduling policy, provided that the stage definition has been done with taking the latter into account.

## 3 Problem formulation and proposed methods

We consider the case of a single-CPU database server with a memory-resident workload. Each query submitted to the server passes through several stages of execution, and each one of those stages corresponds to a server *module*. A module is defined as a relatively autonomous, in terms of the data structures owned and accessed, part of the work that the database server performs in response to a request. An example of such a module is the parser or the optimizer of the database (shown in Figure 2.). Data structures referenced and code executed by two different queries during the execution of a single module can overlap by a variable amount of instructions and data. For instance, two different queries may use different execution operators, produce their own plans, and access different tuples; however, they will reference common code such as buffer pool code or index look-up, and common data such as the symbol table or the database catalog.

Once the common data structures and instructions of a module are accessed and loaded in the cache, subsequent executions of different requests within the same module will significantly reduce memory delays and will reduce the cycles-per-instruction (CPI) rate. The model assumes, without loss of generality,

that the entire set of a module's data structures can fit in the higher levels of the memory hierarchy, and that a total eviction takes place when the CPU switches to a different module. The results drawn using this model may easily be transferred from the processor/cache/memory to the CPU/memory/disk hierarchy.

Processor sharing (PS) fails to reuse cache contents, since it switches from query to query in a random way with respect to the query's current execution module. To exploit the data locality, we propose techniques for scheduling queries between different modules, and evaluate them against processor sharing. In order to compare the different scheduling policies we assume a Poisson stream of queries, whereas the query service time follows an exponential distribution. Whenever a query starts execution at a certain module and the common data structures are not already in the cache (that is, the CPU was previously working on a different module), then the query is charged with an additional fixed CPU demand for that module. This extra CPU demand represents the time spent in memory stalls due to fetching common data structures from main memory to cache, for a given query. Under PS, these stalls are the default ones, and herein lies the opportunity for better exploiting data locality.

Clearly, several of the assumptions related to the system architecture (single CPU, in-memory DBMS, context switch costs, common data structure sizes and cache behavior) are simplifications used to understand the problem and create the framework for evaluating the scheduling policies. Relaxing these assumptions does not affect the generality of our results, as is discussed in more detail in Section 6. The assumption of exponentiality with respect to query sizes and interarrival times does not match workloads universally, but it provides a tractable model for comparing the different policies. For the specific problem under consideration, bursty arrivals will only increase the potential of cache-conscious scheduling, and thus, the use of Poisson arrivals is adequate for qualitatively comparing the different scheduling policies. Nevertheless, the experimentation section departs from exponentially distributed query sizes, and examines what would happen in the case of a highly variable distribution (where a small fraction of the workload consists of really large queries). It has been shown [11] that such a distribution can closely model real workloads. Given the assumptions mentioned, the exact problem definition follows.

### 3.1 Problem definition

Queries arrive at a DBMS server according to a Poisson process with rate $\lambda$. Each query passes, always in the same order, through a series of *M* modules. Each module has its own separate queue. There is only one CPU at the system. Whenever a query arrives at the server, it's service time is drawn from an exponential distribution with mean *m*; when that query executes at a given module *i*, it spends time $m_i$. If the immediately previous query execution happened at a different module, then the CPU also spends time $l_i$ (fixed) to

load module *i* common contents in the cache. The goal is to devise a scheduling policy that minimizes the average query response time.

**TABLE 1: Symbol definitions**

| symbol | explanation | value |
|---|---|---|
| $M$ | number of modules | to be set at the experimentation section |
| $\alpha_i, \sum_{i=1}^{M} \alpha_i = 1$ | fraction of a query's total execution time spent at module $i$ | |
| $\lambda$ | query arrival rate | |
| $m$ | mean query service time, when all common data+code is found in cache | |
| $l$ | total time a query spends loading in cache common data+code | |
| $m_i$ | mean query service time, at module $i$, when common data+code is found in cache | $\alpha_i \times m$ |
| $l_i$ | time a query spends at module $i$, loading in cache common data+code | $\alpha_i \times l$ |
| $c\%$ | percentage of query execution time spent on average servicing common instruction and data cache misses | $\frac{l}{l+m}$ |
| $\rho$ | system load | $\lambda \times (m+l)$ |

The baseline scheduling policies are First Come First Serve (FCFS) and the prevailing one, PS. Under FCFS, whenever a query arrives at module 1 it executes and then continues with module 2, until it has executed all M modules. New queries that arrive at module 1 will have to wait until the current query exits the system. Since there is only one CPU, the only queue that actually accumulates input is one at the entrance of module 1. Like PS, FCFS is also oblivious to the module structure and is not locality-aware (each newly loaded module wipes the previous one from the cache).

## 3.2 Proposed scheduling policies

**D-gated** *(Dynamic-gated)*. This policy dynamically imposes a gate on the incoming queries, and executes the admitted group of queries as a batch at each module, until their completion. Execution takes place in a first-come first-served basis at the queue of each module.When the first query in the queue of module 1 starts execution, it fetches the common data structures of the module in the cache. The rest of the queries that form the current batch pass through module 1 without paying the penalty of loading the common data structures in the cache. When the last query of the batch finishes execution at module 1, the CPU shifts to

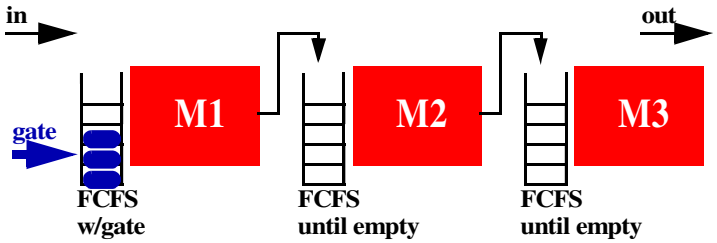


**FIGURE 3:** *Illustration of dynamic - gated for three modules*

the queue of module 2 and again processes the whole batch in a FCFS fashion. Meanwhile, incoming queries to the database server are queued up, at the first module's queue. Eventually, the current batch moves to the last module, and each query leaves the system immediately after its execution. Then, the CPU shifts to the first module and marks the new batch of admitted queries. These are the queries that have accumulated so far in the first module's queue. From that point of time and on, a gate is imposed to all incoming queries, for the duration of the next batch's execution. Since the gate defines a batch size each time module 1 resumes execution, we call this policy Dynamic-gated, or *D-gated*. The whole process is illustrated in Figure 3 for three modules. Whenever the CPU shifts to module 1 and the queue is empty, D-gated reduces to plain FCFS. Note that D-gated is a cache-conscious scheduling policy since it only pays the penalty of loading a module in the cache once per batch of queries.

**T-gated(*N*)** *(Threshold-gated)*. This policy works similarly to D-gated, except for the way it specifies the size of the admitted batch of queries. T-gated explicitly defines an upper threshold *N* for the number of queries that will pass through module 1 and form a batch of maximum size *N*. If more than *N* queries have queued up in module 1 when the CPU finishes with the previous batch, T-gated will admit just *N* queries while the rest will be considered for the next batch. For *N*=1, this policy reduces to FCFS.

**non-gated**. This policy admits all queries queued up in module 1, and works on module 1 until the queue becomes empty. At that point the CPU moves to the next module and proceeds in the same fashion as D-gated and T-gated. Once the current batch exits the system, the CPU shifts to module 1 and keeps admitting queries until there is no more work to be done at module 1. *Non-gated* is more sensitive to starvation, since a continuous stream of queries might cause the server to work indefinitely on module 1. While the analytic workloads we used didn't produce this behavior, the use of a time-out mechanism is necessary in a real implementation. This policy is similar to the one described in [1] (when applied to our framework).

**C-gated** *(Cutoff-gated)*. One possible issue with the two previous policies is that a very large query can essentially block the way to other, smaller ones, and thus lead momentarily to higher response times. With an exponential distribution of query sizes, this problem does not surface when the average response

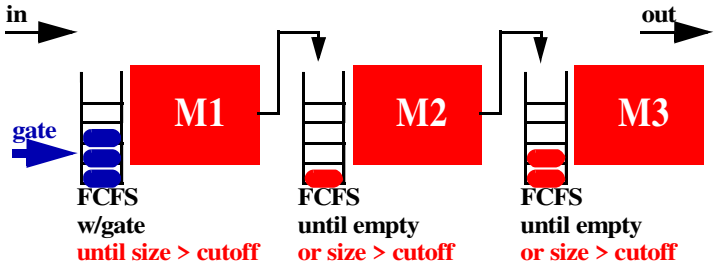


**FIGURE 4:** *Illustration of cutoff-gated for three modules*

time is measured. This is because the majority of the system load is attributed to relatively small query sizes (close to the mean). A heavy-tailed distribution on the other hand, typically involves half the system load to be made up by infrequent but very large queries. In a scenario like that, FCFS based policies could lead to unreasonably high response times, compared to processor-sharing. C-gated tries to bridge the cache-awareness that D-gated and T-gated exhibit with the fairness in the presence of large queries that PS shows. Under C-gated, apart from the imposed gate to the incoming queries at the first module (either dynamically or as a predefined threshold), an additional cutoff value applies to the time the CPU spends on a given query at a given module. Whenever the CPU exceeds that cutoff value, it switches execution to the rest of the queries in the queue and to the next module, leaving the large query unfinished. That query will rejoin the next batch, and eventually resume execution. Whenever its remaining CPU demand for the current module drops below the cutoff value, it will advance to the next module. This way, both small and large queries make progress while still benefiting from increased data locality. The cutoff technique resembles the Foreground-Background scheduling policy used in Unix, where large jobs, when identified, are pushed in a separate queue and receive service only when there are no small jobs in the default queue. The difference is that under C-gated large queries receive service at the presence of small queries, once per batch passing.

### 3.3 Analysis of PS and FCFS

Under both FCFS and PS each query sees all modules as if they were one M/G/1 server, with mean service time the mean module service time plus the module load time. That is, each query has to serve for $l$ time units plus a variable amount of time drawn from an exponential distribution with mean $m$. For PS, the mean response time (expected time in system) in a M/G/1 server, is:

$$E[T_s] = \frac{1}{\frac{1}{m+l} - \lambda} \quad (PS)$$

For FCFS, the Pollaczek - Khinchin formula applies to the expected time in system for a M/G/1 server:

$$E[T_Q] = \frac{\rho}{1-\rho} \cdot \frac{E[S^2]}{2E[S]} \quad (P\text{-}K)$$

This formula needs the first two moments of the general query size distribution (Exp($m$) + $l$):

$$E[S] = \int_l^\infty xf(x)dx = l + m \quad (1),$$

$$E[S^2] = \int_l^\infty x^2 f(x)dx = l^2 + 2lm + 2m^2 \quad (2)$$

So, the expected response time for a query for the given problem definition, under FCFS, is:

$$E[T_s] = \frac{\lambda(l^2 + 2lm + 2m^2)}{2(1-\rho)} + l + m \quad (FCFS)$$

# 4 Analysis of the M/M/1 queue with a staged, locality-aware policy

This section considers a special case of the M/M/1 queue where the notion of data locality is incorporated into the job service times. While in CPU, we assume that each job undergoes a series of execution steps (modules). At each of those modules the service time is affected by the cache hit ratio. Whenever a job moves to the next execution module, new data structures and new instructions need to be loaded in the cache. However, if a second job is preemptively scheduled to execute the very same module that the previous job was working on, then, the cache hit ratio increases, and thus, the service time for the second job is reduced.

Although the analysis described here is in the context of the staged server paradigm, it can also apply to a wider class of servers that follow a "production-line" model of operation. For instance, we can imagine a single robotic arm (could be part of a chain) performing several tasks on incoming items. Suppose some or all of those tasks require each time a special, time-consuming preparation on behalf of the robotic arm (such as switching position or functionality). Then, it could be more efficient if the robotic arm treated incoming items as a batch and performed each task on the whole batch (thus paying the penalty of preparation only once per incoming batch). In that system, the notion of the cache and common data structures is replaced by the preparation of the robotic arm that is common to each task.

In order to model this queue, we assume that all jobs pass through the same execution modules. The breakdown of the service requirement into modules is such that the common data associated with each module fit entirely in the cache (or in the higher levels of the memory hierarchy, in general). The total service time for a job that always suffers the default cache miss ratio (as in the FCFS case) is drawn from an exponential distribution with mean $1/\mu$. A job that always executes a module just after another job has brought the common data of that module in the cache, requires a total service time drawn from an exponential distribution with mean $c/\mu$, with $0 < c \leq 1$.

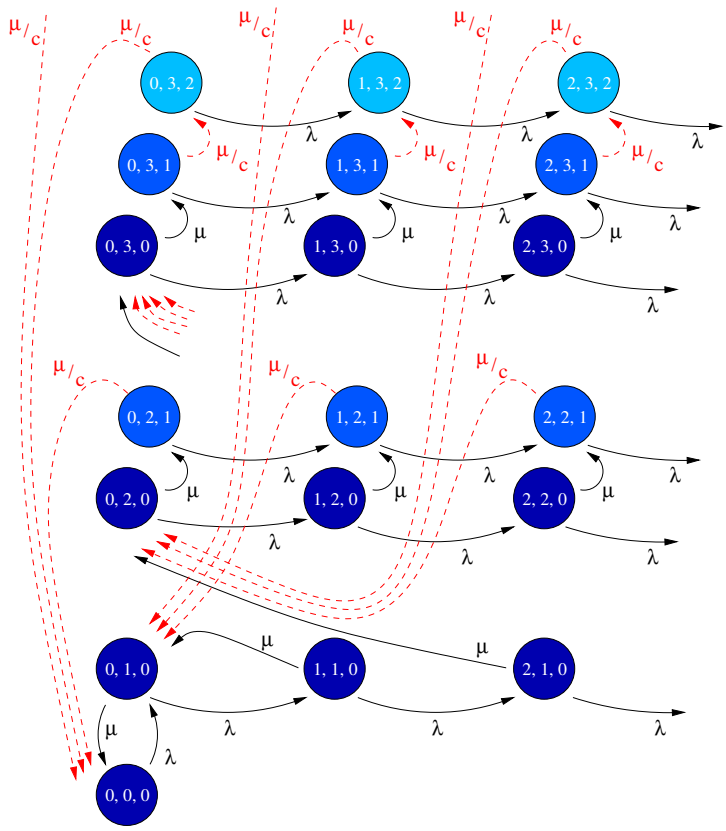


**FIGURE 5:** *Markov chain with states: (jobs in queue, size of admitted batch, jobs completed within batch)*

All existing scheduling policies (preemptive and non-preemptive) for the M/M/1 queue are oblivious to cache performance. For exponential service times all non size-based policies result in the same expected time in system, namely:

$$E[T_s] = \frac{1}{\mu - \lambda},$$

where $\lambda$ is the job arrival rate. The rest of this Section analyses d-gated for the M/M/1 queue just described.

An equivalent way to think of the proposed cache-conscious execution is as a modified FCFS policy. Under this policy the first job in the queue (and in a group) gets uninterrupted service with rate $\mu$ and then waits until all jobs in its group finish. Each of those jobs in turn, executes in a FCFS fashion, but with an "accelerated" rate of $\mu/c$, and then waits for the rest (note that $c$ was defined to be between zero and one). When the last job of the group finishes execution, all jobs leave the system at the same time and the CPU looks at the queue: the jobs that wait there will consist the next group (if there are no jobs in the queue, then the first job to arrive in the system will consist a group by itself, in which case the cache-conscious execution reduces to a true FCFS policy). This modified FCFS scheme behaves the same as the proposed cache-conscious execution in terms of the number of jobs in the system at any time.

Since both the interarrival and service times are exponential, a Markov chain can be written for this modified FCFS scheme (illustrated in Figure 5). A state (*m, n, z*) in this chain is defined as: *m* is the number of jobs waiting in the queue (not admitted), *n* is the number of jobs consisting the current batch, and, *z* is the number of jobs, out of the current batch, that would have finished execution under normal FCFS but now have to wait for all members of the batch to complete execution. Every batch comes to completion whenever there is a departure with rate $\mu$ (for batch size = 1) or rate $\mu/c$ (for batch size > 1), from a state in the form of: (*m, n, n-1)*; this departure leads to state (*0, m, 0*), since all *m* jobs waiting in the queue will consist the next batch.

**Solution of Markov chain.** We can write a balance equation for the state (*m, n, z*): the rate we leave that state equals the rate at which we enter into that state.

$$\left(\frac{\mu}{c}+\lambda\right)P_{m,n,z} = \frac{\mu}{c}P_{m,n,z-1} + \lambda P_{m-1,n,z} \text{ , for } 1 < z < n \quad (1)$$

For $z = 0$, the balance equation is:

$$(\mu+\lambda)P_{m,n,0} = \lambda P_{m-1,n,0} \rightarrow P_{m,n,0} = \left(\frac{\lambda}{\mu+\lambda}\right)^m P_{0,n,0} \text{ , } (2)$$

*Convention: from now on we will write all probabilities of the form* $P_{0,n,0}$, *as* $P_n$*..The goal is to express all state probabilities as functions of* $P_n$.

For $z = 1$, the balance equation is:

$$\left(\frac{\mu}{c}+\lambda\right)P_{m,n,1} = \mu P_{m,n,0} + \lambda P_{m-1,n,1} \text{ , } (3)$$

By substituting (2) into (3) and solving the recurrence, we derive:

$$P_{m,n,1} = \frac{\mu}{\mu/c+\lambda} \cdot \left( \sum_{i=0}^{m} \left(\frac{\lambda}{\mu/c+\lambda}\right)^i \cdot \left(\frac{\lambda}{\mu+\lambda}\right)^{m-i} \right) \cdot P_n \text{ , } (4)$$

For $z = 2$, (1) and (4) give, after solving for the recurrence:

$$P_{m,n,2} = \frac{\mu/c}{\mu/c+\lambda} \cdot \frac{\mu}{\mu/c+\lambda} \cdot \left( \sum_{i=0}^{m} \left(\frac{\lambda}{\mu/c+\lambda}\right)^i \cdot \left(\frac{\lambda}{\mu+\lambda}\right)^{m-i} + \ldots + \sum_{i=m}^{m} \left(\frac{\lambda}{\mu/c+\lambda}\right)^i \cdot \left(\frac{\lambda}{\mu+\lambda}\right)^{m-i} \right) \cdot P_n \text{ , } (5)$$

Similarly, for $z = 3$, we have:

$$P_{m,n,3} = \left(\frac{\mu/c}{\mu/c+\lambda}\right)^2 \cdot \frac{\mu}{\mu/c+\lambda} \cdot \left( \sum_{i_3=0}^{m} \sum_{i_2=i_3}^{m} \sum_{i_1=i_2}^{m} \left(\frac{\lambda}{\mu/c+\lambda}\right)^{i_1} \cdot \left(\frac{\lambda}{\mu+\lambda}\right)^{m-i_1} \right) \cdot P_n \text{ , } (6)$$

From (2), (4), (5), and (6) we can conclude what the formula for $P_{m,n,z}$ as a function of $P_n$ is.

$$P_{m,n,z} = \left(\frac{\mu/c}{\mu/c+\lambda}\right)^{z-1} \cdot \frac{\mu}{\mu/c+\lambda} \cdot \left( \sum_{i_z=0}^{m} \sum_{i_{z-1}=i_z}^{m} \cdots \sum_{i_1=i_2}^{m} \left(\frac{\lambda}{\mu/c+\lambda}\right)^{i_1} \cdot \left(\frac{\lambda}{\mu+\lambda}\right)^{m-i_1} \right) \cdot P_n \qquad 0 \le m, 0 < z < n$$

$$P_{m,n,0} = \left(\frac{\lambda}{\mu+\lambda}\right)^{m} P_n \qquad 0 \le m, 0 < n \qquad , \quad (7)$$

*Convention: from now on, a[m, z] will be the following sum*

$$a[m,z] = \sum_{i_z=0}^{m} \sum_{i_{z-1}=i_z}^{m} \cdots \sum_{i_1=i_2}^{m} \left(\frac{\lambda}{\mu/c+\lambda}\right)^{i_1} \cdot \left(\frac{\lambda}{\mu+\lambda}\right)^{m-i_1} \quad , \quad (8)$$

*a[m, z]* can be thought as a sequence for $z > 0$ with $m$ being a non negative integer, and the following recursive definition can be written:

$$a[m,z] = \sum_{k=0}^{m} \left[ \left(\frac{\lambda}{\mu/c+\lambda}\right)^{k} \cdot a[m-k, z-1] \right] \qquad 0 \le m, 1 < z$$

$$a[m,1] = \sum_{k=0}^{m} \left[ \left(\frac{\lambda}{\mu/c+\lambda}\right)^{k} \cdot \left(\frac{\lambda}{\mu+\lambda}\right)^{m-k} \right] \qquad 0 \le m \qquad , \quad (9)$$

$$a[m,0] = \left(\frac{\lambda}{\mu+\lambda}\right)^{m} \qquad 0 \le m$$

This will be useful when we want to evaluate those probabilities. We now need to compute probabilities $P_n$. For $n > 1$, we have the following balance equation:

$$(\mu+\lambda) \cdot P_n = \mu P_{n,1,0} + \sum_{i=2}^{\infty} \frac{\mu}{c} P_{n,i,i-1} \quad ,$$

and by using (7) and (9), we can rewrite that as:

$$P_n = \frac{\mu \cdot \lambda^n}{(\mu+\lambda)^{n+1}} \cdot P_1 + \frac{\mu}{\mu+\lambda} \cdot \sum_{i=2}^{\infty} \left[ a[n, i-1] \cdot \left(\frac{\mu/c}{\mu/c+\lambda}\right)^{i-1} \cdot P_i \right] \qquad 2 \le n \qquad , \quad (10)$$

For $n = 1$, we can write the following balance equation:

$$(\mu+\lambda) \cdot P_1 = \lambda P_0 + \mu P_{1,1,0} + \sum_{i=2}^{\infty} \frac{\mu}{c} P_{1,i,i-1} \quad ,$$

which can be written as:

$$P_1 = \frac{\lambda}{\mu+\lambda} \cdot P_0 + \frac{\mu \cdot \lambda}{(\mu+\lambda)^2} \cdot P_1 + \frac{\mu}{\mu+\lambda} \cdot \sum_{i=2}^{\infty} \left[ a[1, i-1] \cdot \left(\frac{\mu/c}{\mu/c+\lambda}\right)^{i-1} \cdot P_i \right] \qquad , \quad (11)$$

The probability of being in state 0 (system being idle) equals 1 minus the sum of the probabilities of being in every other possible state, namely:

$$P_0 = 1 - \sum_{m=0}^{\infty} \sum_{n=1}^{\infty} \sum_{z=0}^{n-1} P_{m,n,z} \ ,$$

or:

$$P_0 = 1 - \sum_{n=1}^{\infty} \sum_{m=0}^{\infty} \sum_{z=0}^{n-1} \left[ \left( \frac{\mu/c}{\mu/c+\lambda} \right)^{z-1} \cdot \frac{\mu}{\mu/c+\lambda} \cdot a[m,z] \cdot P_n \right] \ , \quad (12)$$

From equations (10), (11), and (12) we can see that the probabilities $P_1$ through $P_\infty$ are part of a set of an infinite number of linear equations, and their value could had been obtained if we were able to solve this linear system of infinite equations. Such a solution would have the following general representation:

$$\begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ \dots \\ P_n \\ \dots \end{bmatrix} = \begin{bmatrix} \frac{\mu\cdot\lambda}{(\mu+\lambda)^2} - 1 - S(1) & \frac{\mu}{\mu+\lambda}\cdot\frac{\mu/c}{\mu/c+\lambda}\cdot a[1,1] - S(2) & \dots & \frac{\mu}{\mu+\lambda}\cdot\left(\frac{\mu/c}{\mu/c+\lambda}\right)^{n-1}\cdot a[1,n-1] - S(n) & \dots \\ \frac{\mu\cdot\lambda^2}{(\mu+\lambda)^3} & \frac{\mu}{\mu+\lambda}\cdot\frac{\mu/c}{\mu/c+\lambda}\cdot a[2,1] - 1 & \dots & \frac{\mu}{\mu+\lambda}\cdot\left(\frac{\mu/c}{\mu/c+\lambda}\right)^{n-1}\cdot a[2,n-1] & \dots \\ \frac{\mu\cdot\lambda^3}{(\mu+\lambda)^4} & \frac{\mu}{\mu+\lambda}\cdot\frac{\mu/c}{\mu/c+\lambda}\cdot a[3,1] & \dots & \frac{\mu}{\mu+\lambda}\cdot\left(\frac{\mu/c}{\mu/c+\lambda}\right)^{n-1}\cdot a[3,n-1] & \dots \\ \dots & \dots & \dots & \dots & \dots \\ \frac{\mu\cdot\lambda^n}{(\mu+\lambda)^{n+1}} & \frac{\mu}{\mu+\lambda}\cdot\frac{\mu/c}{\mu/c+\lambda}\cdot a[n,1] & \dots & \frac{\mu}{\mu+\lambda}\cdot\left(\frac{\mu/c}{\mu/c+\lambda}\right)^{n-1}\cdot a[n,n-1] - 1 & \dots \\ \dots & \dots & \dots & \dots & \dots \end{bmatrix}^{-1} \cdot \begin{bmatrix} -\frac{\lambda}{\mu+\lambda} \\ 0 \\ 0 \\ \dots \\ 0 \\ \dots \end{bmatrix}$$

where the quantity *S(n)* is defined as:

$$S(n) = \sum_{m=0}^{\infty} \sum_{z=0}^{n-1} \left[ \left( \frac{\mu/c}{\mu/c+\lambda} \right)^{z-1} \cdot \frac{\mu}{\mu/c+\lambda} \cdot a[m,z] \right]$$

**Evaluation.** The above solution can be approximated by solving for a finite *n*. Since the last job on every batch sees the system as if it was a simple FCFS server, the system is stable and the sum of the state probabilities goes to 1 as *n* goes to infinity. We used dynamic programming to efficiently evaluate *a[m,z]*, and matlab to solve the linear equations, for *n* in the range of 100-500. The results matched exactly the simulation scripts, for all combinations of $c, \mu, \lambda$ tried.

## 5 Experiments

In all of our experiments, we set *M*, the number of modules, equal to five. For simplicity, an equal percentage of service time breakdown is assigned to the five different modules (that is, a query spends equal time in all modules or $a_i = 0.2$, for all *i*). PS and FCFS are not affected by the number of modules neither the service time breakdown. The gated algorithms can actually benefit by a biased assignment of service times to the different modules. This happens when queries spend a significant amount of time at the last module.

On average, those queries will leave the server faster since they execute mostly in a FCFS fashion (and thus, are not delayed by all queries in the batch) and benefit from module locality, at the same time. Since this is not typically the case in a real DBMS, the number of modules and service time breakdown remain the same through all experiments.

The time it takes a module to load the common data structures and instructions, $l_i$, is equal for all queries and it varies in the experiments as a percentage of the total expected (mean value) execution time of a query. The mean query execution time (CPU demand) for the case where all common instructions and data structures need to be loaded in the cache (which always is the case for FCFS and PS) is 100ms. This value consists of a fixed component (*l*, time it takes all modules to load common data and code), and a variable one (*m*, query service time when all common code and data is found in the cache). The variable component takes values in the experiments from two distributions: the exponential and the bounded pareto (a highly variable distribution). Since other values for the total mean CPU demand resulted in the same relative differences in response times among all policies tested, all performance graphs are based only on that value. The results for both PS and FCFS are derived from analytical formulas, while all gated policies are based on simulation scripts. The confidence intervals were tight enough and so they are omitted from the graphs, for better readability. Table 2 shows all the experimentation parameters, along with their value range.

**TABLE 2: simulation parameters**

| parameter | variance | value range |
|---|---|---|
| *M*, number of modules | fixed | 5 |
| $a_i$, fraction of execution time at module *i* | fixed | 0.2 |
| $\lambda$, query arrival rate | Poisson | 0-12 queries/sec |
| *m*+*l*, query service time, no module loaded | see below for *m*, *l* | mean = 100ms |
| *m*, query service time, all modules loaded | exponential, bounded pareto | mean = 0-100% of 100ms |
| *l,* common data+code loading time | equal for all queries | 0-100% of 100ms |

## 5.1 Effect of various degrees of data locality (different module loading times - *l*)

In the first experiment the variable component of the mean query CPU demand is drawn from an exponential distribution with mean in the range of 40-100ms. Initially, the query arrival rate is set to produce a system load of 80%. This experiment compares the mean response time for a query under PS, FCFS, D-gated, non-gated and T-gated(2), for various module loading times (the time it takes all modules to fetch the common data structures and code in the cache, *l*). This time varies as a percentage of the mean query CPU demand, from 0% to 60% (the mean query service time that corresponds to private data and instructions, *m*, is adjusted accordingly so that *m*+*l*=100ms). This value (*l*) can also be viewed as the percentage of exe-

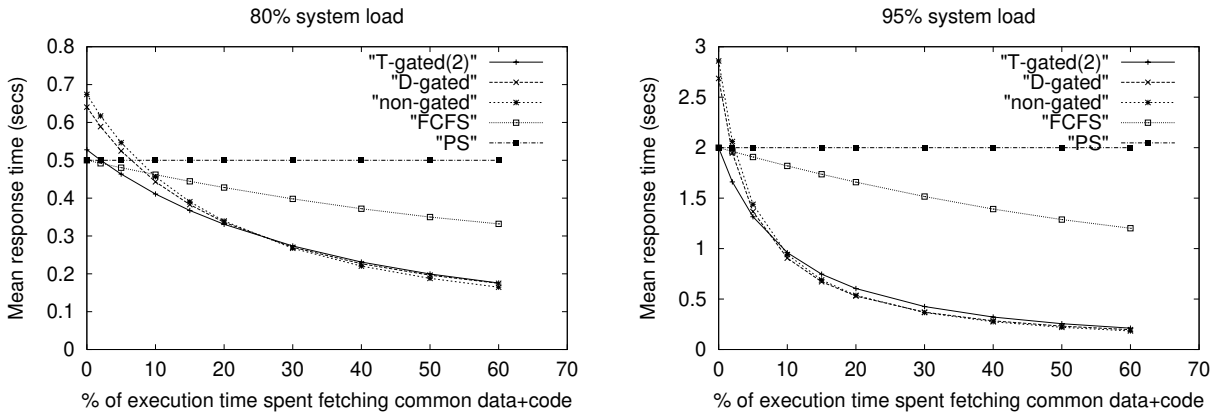


**FIGURE 6:** *Mean response times for 80% system load (left) and 95% system load (right)*

cution time spent servicing cache misses, attributed to common instructions and data, under the default server configuration (e.g. using PS). The results are in Figure 6.

The left graph of Figure 6 shows that the gated family of algorithms performs better than PS for module loading times that account for more than 8% of the query execution time. Response times are up to twice as fast and improve as module load time becomes more significant. On the other hand, for module loading times that correspond to less than 8% of the execution time, non-gated and D-gated policies show up to 20% worse response times. Among those two gated policies, D-gated performs best. T-gated(2) performs consistently well, outperforming PS in almost all configurations. The reason that T-gated(2) performs better than D-gated or non-gated is because it closer approximates FCFS than D-gated does and thus, the first query of every batch of two queries is delayed by only one other query. When the benefits of cache hits are reduced, it is more important for a query not to be delayed by other queries. Note that the response time under FCFS drops as the percentage of module loading time increases. This is because a fixed part in the service time reduces variability and thus, queueing time delays. T-gated performed better in this scenario for a threshold value of $N = 2$.

For the right graph in Figure 6, the same experimental setup is used, but the arrival rate is set to create a system load of 95%. As the system load increases, the trends in the gated family of policies do not change. The performance of PS and FCFS though drops significantly and as a result, gated policies perform better for almost all values of module loading time percentages (> 2%). The gains of the gated policies now increase to up to 7 times reduced response times. T-gated(2) is still the policy of choice, since it never loses to PS and its ability of improving the performance (when compared to PS) even when the common memory references are low, outweigh its slightly worse performance than D-gated and non-gated for high percentages of common memory references.

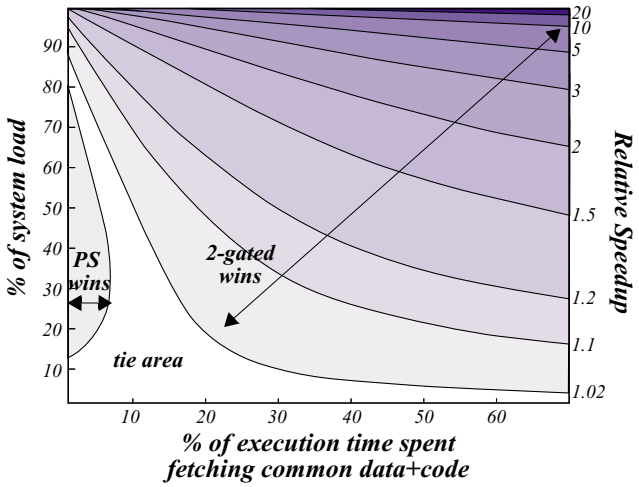


**FIGURE 7:** *Direct comparison of PS and T-gated(2)*

## 5.2 Direct comparison of T-gated(2) and PS

This experiment directly compares PS and T-gated(2) (T-gated with $N$ set to 2) by plotting the area that each of those policies results in lower response times. The same exponential distribution is used for the query sizes. The graph in Figure 7 is produced by varying both the system load and the module loading times. It shows the relative speedup of T-gated(2) over PS, for a wide range of different locality scenarios. The x-axis is the percentage of execution time that is eliminated for a query that finds the common data structures of a module in the cache. This value varies from 1% to 70%. The y-axis is the server load; we varied the arrival rate to achieve server loads between 1% and 98%. On the right of the y-axis we denote the areas where the relative speedup of T-gated(2) over PS is within a certain range. Areas with darker color correspond to higher speedup, while the white area corresponds to those combinations that both policies perform almost the same. PS is only able to perform better than T-gated(2) in a small area on the left of the graph; the relative speedup of PS in that area does not exceed 1.1.

## 5.3 Effect of very large query sizes

For our last experiment, we study the effect of a highly variable distribution. The exponential distribution assumes that the remaining query service times are independent of the CPU time they used so far. This is not true in many real workloads, where many small queries are often the case, while the few large ones are increasingly more likely to take more of the CPU time. Exactly this property (also called decreasing failure rate) is characteristic of a heavy-tailed distribution. Under such a distribution (one example being the pareto distribution), a very small fraction of the largest queries can comprise as much as half of the system load. While PS, as a fair policy, is insensitive to different distributions (we only need the mean service time and the arrival rate to compute the mean response time), the rest of the policies are not. A really large query

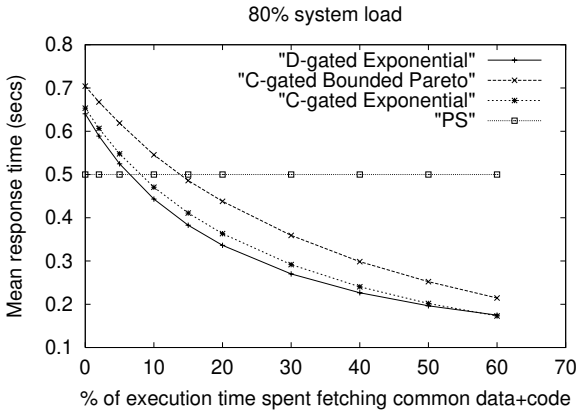


**FIGURE 8:** *Highly variable distribution: Bounded Pareto*

can block the way of many smaller ones and thus, result in higher mean response times. In this section we test how well C-gated can push small queries out of the system fast, while still exploiting data locality.

We use the same distribution as in [11], Bounded Pareto, *B(k, p, a)*, for the variable component of the total query size, with fixed mean in the range of 40-100ms (depending of the choice for *l*, so that the mean query CPU demand remains 100ms). Parameter *p* denotes the maximum query size; we set this to 10,000sec. Parameter *k* is the smallest query size and was set to 12.2ms (its value is computed from the rest parameters when we set the mean). Parameter *a* typically ranges between 0 and 2, and defines the degree of variability (the same way as in Pareto). As *a* approaches zero, the distribution becomes more variable. Bounded Pareto differs from Pareto in that it has all of its moments finite, but it still produces query sizes with very high variance. We chose to set *a* equal 1.1. The probability mass function for the Bounded Pareto *B(k, p, a)* is defined as:

$$f(x) = \frac{ak^a}{1-(k/p)^a} \cdot x^{-a-1} \qquad k \le x \le p$$

For the C-gated policy we manually set a fixed cutoff value of 100ms (same as the mean CPU demand). In a real environment, this can be implemented by monitoring the query sizes and using the average value as the cutoff. We repeated the first experiment (Figure 6) but this time we tested C-gated under both the exponential and the Bounded Pareto distributions. FCFS, non-gated, D-gated, and T-gated explode under Bounded Pareto (they result into very large response times) and thus they don't appear in the plot. Instead, we show again D-gated response times under the exponential distribution. PS, by definition, behaves exactly the same under both distributions. The results are in Figure 8. For the case of C-gated under Bounded Pareto, the confidence intervals are also shown (here they are wider due to the highly variable distribution).

As Figure 8 shows, C-gated under Bounded Pareto wins over PS for module loading times of 15% or more of the mean query execution time. While C-gated under Bounded Pareto is worse when compared to D-gated under the exponential distribution, it nevertheless manages to exploit data locality through cache-conscious scheduling and avoid the pitfall of working almost indefinitely on very large queries. What is interesting is that C-gated under the exponential distribution is only slightly worse than D-gated. This means that C-gated could be the policy of choice when there is no apriori knowledge of the workload characteristics. By increasing the system load, the same trends as in the right graph of Figure 6 are observed. That is, the gains of the gated family over PS are increased.

# 6 Relaxing the model's assumptions

In order to approach the problem of low cache performance in today's servers and particularly DBMSs, due to uncontrolled in-and-out swapping of concurrent requests at the CPU, several assumptions were made. This served the purpose of formulating a framework under which existing and proposed solutions could be compared and understood. This section discusses how the assumptions related to the architecture of the system affect the overall solution and the experimental results:

**Single CPU.** High-end commercial database system installations typically run on high-performance multiprocessor systems. Although this essentially adds a degree of freedom to the space of possible scheduling policies, it is straightforward to extend the proposed approach to include execution on multiple CPUs. We divide the modules into groups, as we assign each group to a processor. The analysis in this report then holds for each individual CPU. Moreover, there is an additional opportunity for cache-conscious execution: by assigning specific modules at each CPU and scheduling the queries accordingly, an even higher number of queries pass through a module per CPU, and thus, higher performance gains are expected (as it was illustrated in Figure 6b, for increased system load).

**In-memory DBMS**. Although the model discussed is based on the processor/cache/memory hierarchy, our results can easily be applied to the CPU/memory/disk hierarchy, exploiting memory locality. I/O interrupts cause the suspension of a thread execution, before the end of the time slice assigned to that thread. Real servers are based on this mechanism to overlap and mask the latencies of the various system devices. The assumption of an in-memory DBMS removes the need of premature thread preemption (except for synchronization purposes), and thus, allows for a simpler execution model. Nevertheless, we found that blocking threads do not affect the way the proposed policies work. Whenever a thread unblocks, it either joins the next batch of queries passing through the module, or it executes alone, if the CPU was idle.

**Context switch cost**. This cost applies to all policies (except first-come-first-serve with no I/O interruptions) and thus it doesn't affect the relative performance gains. Moreover, context switches tend to hap-

pen less frequently for the gated family of policies, since a query tries to execute entirely, without interruptions (if possible) when at a given module.

**Common data structures / cache benefits model.** The proposed framework models the benefit of cache hits due to the staged execution with a single parameter per module (percentage of execution time spent servicing cache misses attributed to: common instructions and code between two queries) that includes all the possible code/data overlap between queries executing in a given module. This number is a combination of all the overlaps in several data categories, and simplifies the presentation of the comparison among scheduling policies.

# 7 Conclusions

Modern servers and especially commercial database servers that run on current platforms typically suffer from high processor/memory data and instruction transfer delays. Despite the ongoing effort to create locality-aware algorithms, the interference caused by context-switching results in high penalties due to additional conflict and compulsory cache misses. To preserve data locality across execution threads, the staged server [1] programming model allows for the incoming queries to queue behind each of the database server modules, and appropriate scheduling policies minimize conflicts across different modules' working sets. This paradigm is ideal for database systems, due to their modular and memory-demanding nature.

The contributions of this technical report are twofold. First, it introduces four novel cohort scheduling techniques for staged software servers that follow a “production-line” model of operation. Next, it presents a mathematical framework to methodically quantify the performance trade-offs when using these techniques. Our Markov chain based analysis can apply to a wide class of systems that may benefit from a stage divided service, repeatedly performed on batches of jobs. In the context of a simulated database system, we compare six scheduling alternatives: the traditional processor-sharing model, plain FCFS, and the four proposed scheduling techniques.

In order to show the performance gains under each possible scenario, we vary parameters such as the importance of the module loading time when compared to the execution time and the overall load in the system. The experimental results show that the new scheduling policies (a) outperform processor-sharing by delivering up to a seven-fold response time improvement in most configurations, and (b) exploit data and instruction locality for a wide range of workload parameter values. Although the expected gains in a real, large staged system can be lower, due to additional, implementation related overheads, this report shows nevertheless, that the scheduling trade-off of delaying the requests forming a batch is justifiable and pays off for even low degrees of inter-request locality.

# 8 References

[1] James Larus and Michael Parkes. “Using Cohort Scheduling to Enhance Server Performance” Microsoft Research Technical Report MSR-TR-2001-39, March 2001.

[2] Shimin Chen, Phillip B. Gibbons and Todd C. Mowry. “Improving Index Performance through Prefetching” Proceedings of the SIGMOD 2001 Conference, May 2001.

[3] T. M. Chilimbi, J. R. Larus and M. D. Hill. “Making Pointer-Based Data Structures Cache Conscious”. IEEE Computer, December 2000.

[4] Goetz Graefe, Per-Åke Larson. “B-Tree Indexes and CPU Caches”. Proceedings of the International Conference on Data Engineering 2001, pp.349-358.

[5] A. Shatdal, C. Kant, and J. Naughton. “Cache Conscious Algorithms for Relational Query Processing”. In proceedings of the 20th International Conference on Very Large Data Bases (VLDB), pp. 510-512, September 1994.

[6] A. Ailamaki, D.J. DeWitt, M.D. Hill, and M. Skounakis. "Weaving Relations for Cache Performance". Proceedings of the 27th International Conference on Very Large Databases (VLDB), Roma, Italy, September 2001

[7] A. Ailamaki, D.J. DeWitt, M.D. Hill, and D.A. Wood. "DBMSs on a modern processor: Where does time go?," Proceedings of the 25th International Conference on Very Large Databases (VLDB), Edinburgh, Scotland, September 1999.

[8] J. Jayasimha and A. Kumar, "Thread-based Cache Analysis of a Modified TPC-C Workload," in Proceedings of the Second Workshop on Computer Architecture Evaluation Using Commercial Workloads. Orlando, FL, 1999.

[9] M. Rosenblum, E. Bugnion, S. A. Herrod, E. Witchel, and A. Gupta, "The Impact of Architectural Trends on Operating System Performance," in Proceedings of the Fifteenth ACM Symposium on Operating System Principles. Copper Mountaint Resort, CO, 1995, pp. 285-298.

[10] K. Keeton, D. A. Patterson, Y. Q. He, R. C. Raphael, and W. E. Baker, “Performance Characterization of a Quad Pentium Pro SMP Using OLTP Workloads,” in Proceedings of the 25th Annual International Symposium on Computer Architecture, Barcelona, Spain, 1998, pp. 15-26.

[11] Mark Crovella, Mor Harchol-Balter, and Cristina Murta, "Task Assignment in a Distributed System: Improving Performance by Unbalancing Load," Proceedings of ACM Sigmetrics '98, Madison, WI.

[12] J. L. Hennessy and D. A. Patterson. “Computer Architecture: A Quantitative Approach,” 2nd edition, Morgan Kaufmann, 1996

[13] Nikhil Bansal and Mor Harchol-Balter. “Analysis of SRPT Scheduling: Investigating Unfairness." To appear in Proceedings of ACM Sigmetrics 2001 Conference on Measurement and Modeling of Computer Systems.

[14] Matt Welsh, David Culler, and Eric Brewer. “SEDA: An Architecture for Well-Conditioned, Scalable Internet Services,” Proceedings of the Eighteenth Symposium on Operating Systems Principles (SOSP-18), Banff, Canada, October 2001.

[15] A. M. G. Maynard, C. M. Donelly, and B. R. Olszewski. Contrasting characteristics and cache performance of technical and multi-user commercial workloads. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, October 1994.

[16] M. S. Squillante and E. D. Lazowska. “Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 2, February 1993

[17] S. Subramaniam and D. L. Eager. “Affinity Scheduling of Unbalanced Workloads”. Proceedings Supercomputing '94, November 14-18, 1994