On-The-Fly Capacity Planning

Nick Mitchell    Peter F. Sweeney
IBM T.J. Watson Research Center
{nickm,pfs}@us.ibm.com

Abstract

When resolving performance problems, a simple histogram of hot call stacks does not cut it, especially given the highly fluid nature of modern deployments. Why bother tuning, when adding a few CPUs via the management console will quickly resolve the problem? The findings of these tools are also presented without any sense of context: e.g. string conversion may be expensive, but only matters if it contributes greatly to the response time of user logins.

Historically, these concerns have been the purview of capacity planning. The power of planners lies in their ability to weigh demand versus capacity, and to do so in terms of the important units of work in the application (such as user logins). Unfortunately, they rely on measurements of rates and latencies, and both quantities are difficult to obtain. Even if possible, when all is said and done, these planners only relate to the code as a black-box: but, why bother adding CPUs, when easy code changes will fix the problem?

We present a way to do planning on-the-fly: with a few call stack samples taken from an already-running system, we predict the benefit of a proposed tuning plan. We accomplish this by simulating the effect of a tuning action upon execution speed and the way it shifts resource demand. To identify existing problems, we show how to generate tuning actions automatically, guided by the desire to maximize speedup without needless expense, and that these generated plans may span resource and code changes. We show that it is possible to infer everything needed from these samples alone: levels of resource demand and the units of work in the application. We evaluate our planner on a suite of microbenchmarks and a suite of 15,000 data sets that come from real applications running in the wild.

1. Introduction

With a bit of planning, the arduous task of optimization can lead to large improvements in maintenance costs and performance. We have seen many situations where a few straightforward changes were enough to tip an application, from the slow death of severe resource contention, to the smooth flow of work through the system. Adding processors to a database machine, splitting the execution of a single-process program into several processes, or updating code to use a concurrent data structure — these changes can yield big reductions in the resources necessary to support anticipated load at desired costs. This paper presents a system that guides the selection and parametrization of tuning actions.

Teams need guidance because, with every turn of the tuning crank, they are again faced with reasoning through how their changes altered the landscape of resource constraints. Did this latest change remove the lock contention bottleneck? If so, then why is performance still bad? Did all of our tuning efforts simply expose another, heretofore latent, bottleneck somewhere else?

Latent Bottlenecks, Zero-sum Games, and Head Fakes

System tuning is often cursed with a richness of possibilities. The best course of action is often not obvious, because the most prevalent activity may not be what you need to fix.

For example, if demand for a critical section is high, hundreds of threads may sit idle, waiting for access to the guarding monitor. If the machine’s CPU is already saturated, then eliminating the lock, at least as an isolated tuning action, may be a futile endeavor. Doing so will only shift resource demand, away from the lock and to an already-saturated CPU. The saturated CPU resource is a latent bottleneck, at least with respect to the prima facie problem, that of hundreds of threads backed up on a lock.

This scenario is also an example of a zero-sum game: those threads already consuming CPU will execute more slowly, due to increased contention for processors, and those that were previously waiting on the lock can now complete more quickly — in equal measure.

Even in the absence of these confounding problems, other kinds of “head fakes” can occur. For example, consider a case with two program tasks $A$ and $B$ competing for a saturated pool of processors. If you are dissatisfied with
the performance of A and happy with that of B, it may, counterintuitively, be easier to address the seemingly less important task, B, first. The right choice depends on which has greater leverage. If B is 10 times as concurrent as A, then, for every doubling of the speed of A, you need only find a way to make B 10% faster.

**The Role of Capacity Planning in Performance Tuning**

This relationship between concurrent demand and resource capacity is at the heart of the field of capacity planning. When done well, capacity planning tools can predict changes to response time and throughput, based on various what-if scenarios, with uncanny precision [6, 7, 10, 14, 16, 21]. For example, a capacity planner might predict that adding two processors to the database machine will result in a 1.3 second decrease in the response time of the user login scenario.

This is valuable information, but comes at a cost of extensive, and largely manual, data collection. In a typical capacity planning exercise, one must provide a list of available resources and the way they are interconnected, specify the queuing semantics of each resource, define the set of program tasks (such as the Servlets), and measure load characteristics such as the arrival rate of work. After this manual data collection, a human often must then babysit the system through a period of curve fitting. These predictive formulae are what enables them to make such precise predictions.

In addition to the user burden, these tools operate in a black box fashion with respect to the code. Despite all of this input data, the best capacity planning tools can say about the code context is the name of the program tasks — and this, only because the users themselves provided it to the tool.

**Our Contribution**  

Figure 1 summarizes the main components of our solution. We first infer properties of resource consumption and resource availability from readily and cheaply available data, collected as the application runs. We then simulate the effect of tuning actions, as they shift demand between resources.

We model the effect of tuning actions upon demand via an abstract simulation that models resources as typed nodes, and concurrent demand for these resources as typed tokens (nicknamed marbles). The type of a resource depends on how it responds to saturation; e.g. a thread pool or Java monitor gates access to a limited number of concurrent consumers, whereas a CPU is multiplexed amongst concurrent demand. The type of a consumer that places demand on a resource (a marble in the model) is given by the program task it is servicing; e.g. user logins and checkout actions would be assigned two distinct marble types.

We model tuning actions as the movement of marbles between nodes. For example, speeding up the critical section of a lock has the effect of shifting a share of demand, away from the lock, and to those resources used in the critical section. We show how to model the benefit of a tuning action by observing the initial and final placement of marbles.

This kind of simulation-driven modeling is similar in spirit to the discrete event simulations employed by some conventional capacity planners [11, 12], except that our events are tuning actions, rather than the arrival of work.

These tuning actions can either be provided by a human, as part of the evaluation of what-if scenarios, or the actions can be provided by the system itself, via the Plan Generator. The plan generator seeks to explain, and provide remedy for, performance problems in the system as it exists now — a resource or an aspect of the code is only part of the problem if it is part of a highly ranked solution. We will show how this automated exploration is guided by the goal of eliminating extant, latent, and looming bottlenecks.

We show that this dual duty, of performance analysis and extrapolatory planning, can be done “on the fly” — i.e. using only easily obtained information from an already-running system. The only information needed is moment-in-time snapshots of thread states, and information about the capacity of resources. This data lacks any kind of temporal information, such as the rate of arrival of work and the latency to complete each piece of work. We show that, rather than being compromised by a paucity of information, on-the-fly planning provides more explanatory detail than either conventional capacity planning systems or hotspot identification tools, without harm to the accuracy of the predictions.
2. Planning: Background and Terminology

Capacity planning is typically seen as the task of identifying the amount of a resource that is necessary to provide quick service to current or anticipated levels of load. For this paper, we consider a plan to be a set of tuning actions, each action applied either to the hardware resources of the system (such as processors), to the software resources of the system (such as thread pools), or to the executing code. On this last point, we also consider an opposite goal of planning: that of reducing load (via code tuning) so that it runs within a given set of resource constraints.

A good plan is one that matches the load on CPU, thread pools, monitors, etc. with the capacity of those resources to execute such load within given time constraints. Under-shooting this crossover point, of supply and demand, usually results in unsatisfied time bounds, and overshooting this “goldilocks” point usually results in unnecessary expenses.

2.1 System Load Becomes Resource Demand

As load arrives for processing it will begin to place demand upon the resources of the system. For example, if single-threaded CPU-intensive tasks arrive at a rate of \( R \) per second, and the CPU resource requires \( L \) seconds to complete the execution this task, a concurrent demand on CPU of \( RL \) will be observed.\(^1\) These two values are termed the arrival rate of load (\( R \) per second) and the service time of a resource (\( L \) seconds of CPU time). If there are brief spikes that double the arrival rate of new tasks, matching brief spikes in concurrent demand can be witnessed.

Realizing resource demand in units of concurrency is important, because most resources can tolerate a certain level of concurrent demand without a degradation in service time. This limit of tolerance for demand is the concurrent capacity of a resource. Continuing our example: if the system has \( RL \) CPUs, then the concurrent demand is well-matched to the concurrent capacity: there will neither be a backlog of requests, nor will there be an excess of CPU capacity. For those brief spikes of arrival rate, service time will degrade, as multiple tasks are multiplexed to the CPUs. However, as long as the long-term average demand does not exceed the product \( RL \), the system can still process the requests in a healthy manner — i.e. without any risk of a catastrophic backlog of requests.

This matching of capacity and demand is an expression of the well-known Little’s Law of queueing theory [15], which we will return to in a later section.

The seminal treatment of queueing network models [14] denotes tasks as customers, and the nature of the work as the “customer class”. In this paper, we use the terminology Work Unit to capture this property, because they represent the unit of accomplishment, and thus the metric of performance. For example, an application might have two Work Units, one for the user logins and one for the shopping cart checkout transaction.\(^2\) The health of the system hinges upon the successful and timely completion of these units of work.

2.2 Making Predictions Knowing Only Demand

This paper introduces a predictive model that does not require any temporal information. Our model can predict the benefit of a tuning plan knowing neither the arrival rate of load nor the service time of the resources. We will show how to do this by observing only levels of concurrent demand and concurrent capacity, and by modeling the effect of tuning plans as a shift of demand between resources.

3. Call Stack and Utilization Samples

For our purposes, a snapshot of the running state of an application consists of a summary of the execution state of its live threads, along with information about overall levels of resource consumption. The information we collect originates from data feeds provided either by the operating system or the application’s runtime environment. This information is readily available, and requires no a priori metrics for throughput, such as transaction rate, nor a built-in mechanism for extracting such metrics.

Most commercial Java Virtual Machines (JVMs) provide a facility for acquiring a snapshot of the execution state of the live threads in a running application. Each snapshot contains the full call stack of each thread, whether awake or idling, in a running application. IBM JVMs call these snapshots “javacores”.

These thread snapshots often also include information about monitors in the running application, in the form of the lock graph. The lock graph is a bipartite graph between monitors and threads. This graph indicates, for each critical section, the current owning thread and the threads that are queued up, waiting to enter that section of code.

The second aspect of information we sample are the operating system resource utilization feeds, such as vmstat and iostat on UNIX platforms.

3.1 Our Data Sets

For the purposes of evaluating this work, we have a rich set of input data from which to draw upon. The data is provided by an existing performance analysis service in wide use. This service provides us with a catalog of over fifteen thousand inputs collected during the execution of a variety of Java batch and server workloads. The data was collected, by users of the service, in a wide variety of settings. For example, users include a plurality of developers, test organizations, service teams responsible for debugging problems in deployed applications, and operations staff triaging issues with the systems they maintain.

\(^{1}\) If you are familiar with the UNIX uptime utility, the load average statistic reports the level of concurrent CPU demand.

\(^{2}\) These two Work Units may or may not emanate from the same customer, in the sense of a registered user of the system.
A risk in using data submitted “from the wild” is that the input call stack samples may be too few in number to accurately represent the true nature of resource consumption. If an application spends on average of 10% of its time consuming a resource, then one would need a minimum of 1431 samples to have a 95% confidence of reflecting exactly that average. Usually, such accuracy is not necessary. If we can tolerate a 5% spread in either direction, i.e. if we can tolerate our samples reporting anywhere from 5%–15% instead of the actual 10%, then we need only 21 samples to have 95% confidence in that spread. If the application spends 20% of its time in a resource, then to have 95% confidence of observing a such a spread, we would need only 11 samples.\footnote{These distributions derive from the Poisson probability function.}

Figure 2 shows a histogram of the number of call stack samples. There are 2598 inputs with at least 10 call stack samples, and 3898 inputs with at least 20 call stack samples.

4. Tuning Actions and Tuning Plans

A tuning action is a change to the code or the environment in which the application runs. For this paper, we consider the goal of a tuning action to be reducing the time required to complete tasks. This is often referred to as service time.

Our Suite of Tuning Actions

A tuning action manipulates the system by either:

1. \textbf{Directly Reducing Service Time}, such as via code optimizations or SQL query tunings that target specific operations or queries. Section 9 shows how to guide the specificity of these actions.
2. \textbf{Adding Capacity}, such as adding CPUs. Section 7 shows that capacity-adding actions are beneficial to the extent that they reduce service time.
3. \textbf{Adding Load}, such as increasing the number of load-driving machines in a testing environment.

A tuning action is also parametrized by a multiplicative tuning factor; e.g. a 2x increase in CPUs.

While not addressed in this paper, changes which move in the opposite direction are also important. We believe that supporting “slowdown” actions, useful when trying to minimize cost-to-benefit ratios, should be straightforward extensions to the content of this paper.

A tuning action makes one alteration. A \textit{tuning plan} is an unordered set of tuning actions, with the implication that their impact on performance be considered as a unit. Section 6 and Section 7 show how to predict the implications of one tuning action. Section 9 extends this to tuning plans.

5. The Marbles Model

The speed of an application can be quantified either in terms of completion throughput or in terms of the time to complete each invocation of a program task. The latter, often called response time in the case of server workloads, is a quantity that must be attached to a specific program task.\footnote{The terminology of the planning community here isn’t ideal: “response time” is the end-to-end time to completion of an entire unit of work. It is also useful to portion out “service times”; e.g. the response time of user logins is dominated by the slow service time of the database queries.}

These program tasks might include the servlets in a Java web application, or the phases in a batch application. Response time is generally expressed in those terms: e.g. user logins complete in 2 seconds. In this paper, we focus on the response time performance metric. We use the term \textit{Work Units} to describe these program tasks, because they are the atomic units of accomplishment in applications.

For now, we will assume that our input data, the call stack samples, explicitly represents two facts that we will later infer: the Work Unit in progress and the resources consumed by that thread, at the moment in time the sample was taken. For example, a thread might be executing the user login Work Unit, and, while servicing this task, be observed to place demand on a CPU and simultaneously own a lock. Section 8 shows how to extract these facts automatically, in the case that they are not explicit.

For testing and validation purposes, we have 15,000 input data sets. This figure shows a histogram of the number of call stack snapshots, across these inputs.

Figure 2. For testing and validation purposes, we have 15,000 input data sets. This figure shows a histogram of the number of call stack snapshots, across these inputs.

Figure 3. A thread that places a unit of concurrent demand against a resource is modeled as a colored “marble” mapped to this resource; a color represents the task being serviced.
We call this relationship between a thread sample and resources: demand. When viewed over all call stacks sampled at a moment in time, this demand relation tells us how much concurrency the application throws at each resource. Four threads observed to place demand upon 8 CPUs at a moment in time indicates a concurrent demand of 4 placed against a concurrent capacity of 8.\(^5\)

We define a model that represents the level of concurrent demand for a set of resources at a moment in time. The unit of concurrent demand is a thread. The resources in this abstraction may either represent local hardware or software resources, or may be proxies for remote resources, in the case that the application runs in a distributed fashion.

**The Marbles Model**

In an instance of a marbles model, each unit of demand is represented by a colored token, called a marble. The color of a marble is the Work Unit in progress in that thread, at the moment the thread sample was acquired. Each marble also has a fractional share that represents the portion of the call stack it represents; its initial value, prior to any tuning, is 1. Each marble is mapped to the resource it demands. A resource has a concurrent capacity, and is either multiplexing (e.g. CPUs) or gating (e.g. locks) in the way it tolerates concurrent demand in excess of concurrent capacity.

\[ \text{Speedup and Residual Demand} \]

Given an application with \( k \) Work Units, \( W_1, \ldots, W_k \), a Tuning Action Simulator must:

1. Compute the speedup factor of the given action upon each Work Unit \( W_i \). We define the speedup of a tuning action on a Work Unit to be the ratio, before versus after tuning, of the expected time to completion of that task.
2. Predict the residual demand that future samples would witness, were they to be taken after the tuning action is applied to the system.

We start by exploring the differences between server and batch workloads, and then show how to bridge these differences with a common simulation model.

### 6.1 The Demand Invariant of Server Workloads

In a server application, work arrives on an ongoing basis. Thus, the amount of demand placed on resources is a function of rate at which work arrives, and the time that the resources require to process each unit of work.

Table 1 steps through the mental exercise of increasing arrival rate without a matching increasing the number of processors. For a while (the first three rows) an increase in arrival rate is tolerated gracefully by the system. In this regime of load, the server runs in a dynamic steady state: arrival rate may have spikes and lulls, but, as long as long-term average rate is not greater than 4 per second, the system can consume load without any backlog.

Observe from the first three rows that, as long as the server remains in a nice steady state, the level of demand for resources is a good predictor of the change in load (which is otherwise hidden from us). Demand is a good predictor of speedup because of an invariance that relates the properties hidden from us to the property, demand, that we can observe.

In a dynamic steady state, the product of arrival rate and service time remains equal to the observed demand, independent of changes in load or service time. This invariant is named Little’s Law\(^6\):

\[ \text{Demand} = \text{Arrival Rate} \times \text{Latency} \quad \text{(Little’s Law)} \]

This law applies both at the granularity of individual resources (where Latency is the service time of the resource), and also at the granularity of a Work Unit (where Latency is the total response time of that Work Unit).

From this, we can predict both speedup and the residual demand that result from a tuning action. For a tuning action that does not affect arrival rate, Little’s Law implies first that speedup, i.e. a ratio of response times, is equal to the ratio of demand; and second, that if a tuning action increases the speed with which a resource can complete each unit of work, future demand for that resource drops by that factor.

For example, if we start out with a concurrent demand of 4 for CPU, and then speed up the operations by a factor of 2, we would witness two effects. First would be a reduction in demand for CPU, from 4 to 2; this tuning action would take us from third row to the second row in Table 1. Second would be a factor of two speedup in response time.

### 6.2 The Demand Invariant for Batch Workloads

For batch workloads, in contrast to server workloads, the total level of concurrent resource demand is unaffected by code becoming quicker. For example, if a part of the batch application is tuned so as to require less CPU, this will not

\(^5\) Our “concurrent capacity” is elsewhere “available parallelism”.

\(^6\) History credits Burton Smith with the restatement, in 1995, of this law in terms relatable to computer science [3]. We use adopt his terminology here.
reduce the amount of *concurrent* demand for CPUs (though it will reduce the total CPU time, of course). To predict the residual demand, it is helpful to know the distribution of demand to resources, for each work unit.

**The Work Unit-Resource Distribution**

For a work unit $W$, the *resource distribution* of $W$ is a probability density function over resources of the likelihood that, during the servicing of $W$, a thread will be observed to be placing demand on a resource.

Say a phase of a batch application is initially observed to place a concurrent demand of 4 threads on CPU, and 6 threads on database resources. The resource distribution for this work unit is (0.4, 0.6). If a tuning action speeds up the CPU portion by a factor of two, this will not reduce total demand from 10 to 8; there is already enough concurrency of the units of work in this phase to feed 10 worker threads, and speeding up an aspect of these Work Units does not change this. After tuning, we have in effect 2 worker threads freed to consume resources in the ratio of 2/8 and 6/8: those freed up worker threads have a 25% chance of being seen consuming CPU and a 75% chance of being seen consuming database resources. We refer to this new distribution (0.25, 0.75) as the *residual* resource distribution.

Thus, those 2 threads would be expected to place 0.5 and 1.5 demand on the two resources, leading to a residual demand of 2.5 to CPU and 7.5 to the database resources.

### 6.3 Modeling Speedup: The Sink Construct

Unlike with a server workload, the speedup of a tuning action in a batch workload is not given simply by the ratio of initial and final demand — the demand invariant of batch workloads keeps the total level of demand constant across tuning actions. Fortunately, there is commonality that we can leverage that will let us predict speedup for these two workloads in a uniform fashion.

Observe that, for a server workload, the factor $f$ to which a tuning action decreases the service time of a resource is the factor by which demand decreases; this is a straightforward reflection of Little’s Law. As a bookkeeping crutch, we can model a decrease in demand as an increase in demand for a special resource that we call the “Sink”. In this way, speedup can be measured by inspection, after applying a number of tuning actions, of the demand for the Sink resource.

#### The Sink Resource Speedup Rule

We introduce a virtual resource called the *Sink*. Any tuning action that reduces the service time of a marble by a factor of $f$ results in $1 - \frac{1}{f}$ of its share flowing to Sink, and $\frac{1}{f}$ remaining as demand for that resource. We model this by *splitting* the marble into two, each with its respective share.

For any workload type, the response time speedup factor to a Work Unit $W$, making sure to count only the demand due to marbles of $W$’s color, is:

![Math equation](image)

For a batch application, we can consider this also to be the case, except that additional load (obeying the residual resource distribution) soaks up any threads made available by that speedup. In order to restore the Batch Demand Invariant, a simulated Add Load action is performed. If $k$ marble shares are moved to Sink, then we add $k$ new marble shares, placed according to the residual work unit-resource distribution. In the previous example, where two units of demand flowed out of CPU and into Sink, and these two units were replaced according to the residual resource distribution of (0.25, 0.75).

### 7. Plan Simulation (When Demand Exceeds Capacity)

With enough load, it is possible for concurrent demand to exceed the concurrent capacity of a resource. For example, in one snapshot, we might observe 5 threads placing demand on 4 CPUs. When demand exceeds capacity, one of three...
situations occurs. Either demand is time-multiplexed to the available capacity, demand queues up waiting for access to available capacity, or performance diverges, as work arrives faster than it can be serviced. We deal with these three situations in turn.

7.1 Multiplexed Resources (CPUs and Network Fabric)
In a server workload, momentary spikes in arrival rate or latency can cause periods of backlog. Backlog results because the amount of concurrent demand exceeds the concurrent capacity of one or more resources. For a server, as long as the long-term average demand for a resource does not exceed its concurrent capacity, brief backlogs will be gracefully tolerated: between the “mountains” of excess, the “valleys” in demand will give the system time to drain pent-up demand.

For a server in a state of brief excess, time to completion degrades linearly, as resources are time-multiplexed between aspiring consumers. If, for a server workload, the long-term average product of arrival rate and service time rises above available concurrent capacity, catastrophic effects ensue; we will return to this shortly.

This multiplexing before concerns us because it opens up the possibility that a tuning action focused on capacity may indirectly result in a decrease in service time.

<table>
<thead>
<tr>
<th>The Speedup Effect of Capacity Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>If a resource with concurrent capacity ( C ) is faced with concurrent demand ( D ), then the multiplexing slowdown is ( \max(1, \frac{D}{C}) ), termed the overcommit factor.</td>
</tr>
<tr>
<td>A tuning action that increases the concurrent capacity of this resource by a factor of ( a ) will cause a reduction in service time by a factor of ( a ), bounded by the overcommit factor.</td>
</tr>
<tr>
<td>Therefore, the effective service time reduction factor of a capacity action of magnitude ( a \geq 1 ) is</td>
</tr>
<tr>
<td>[ f_{\text{capacity}} = \min(a, \max(1, \frac{D}{C})) ]</td>
</tr>
</tbody>
</table>

For example, a 4-way machine with 6 threads demanding access to CPU has an overcommit ratio of 1.5. We can now apply the Sink Resource Speedup Rule from Section 6.3. By plugging in this effective speedup factor, we can compute the expected service time speedup of a capacity action.

7.2 Gating Resources (Monitors and Thread Pools)
The second kind of behavior that may occur when demand exceeds capacity is gating. A gating resource, such as a Java monitor, restricts access to a set of associated resources. They serve as concurrency limiters for the multiplexed resources used in the critical section of the gate. For a monitor, that would be whatever resources are demanded by the block of code executed while the monitor is owned.

All threads that wish to pass through the gate, the pent-up demand, lie idle. For every gating resource, we also associate a queue, and any marbles that represent pent-up demand for the resource are mapped to this queue.

Consider the example shown in Figure 4, which shows three Work Units (the three colors of marbles), and explores increasing the capacity of a gating resource that limits access to a CPU-bound critical section. Observe that, when the CPU has a concurrent capacity of 1, increasing the capacity of the gating resource does not reduce the average service time of the three Work Units. This is an example of a latent bottleneck: the CPU resource is already saturated.

If instead the CPU resource has a capacity of 3, then increasing the capacity of the gate does result in a speedup to the Work Units, but only to the point at which the CPU resource becomes saturated. This is a looming bottleneck.

Latent and looming bottlenecks occur with gating resources due to demand shifting. In the case for a multiplexing resource, a capacity increase may reduce the slowdown due to time sharing. For a gating resource, a capacity increase moves demand: from the pent-up demand queue to the resources of the critical section.

<table>
<thead>
<tr>
<th>The Demand Shifting Effect for Gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let the length of the pent-up demand queue for a gating resource be ( Q ), and the initial concurrent capacity of the gate be ( C ). If a tuning action increases the capacity of a gating resource by a factor of ( a \geq 1 ), then the number of marbles that will shift is ( Ca - C ), but bounded by the length of the queue ( Q ). Thus the fraction of each pent-up marble that shifts is this value divided by ( Q ):</td>
</tr>
<tr>
<td>[ \text{Gate Shifting Fraction} = \min(1, \frac{C(a - 1)}{Q}) ]</td>
</tr>
</tbody>
</table>

When load has shifted to the critical section, this has same effect as an Add Load tuning action on the resources of the critical section. Therefore, when simulating an increase in

![Figure 4](image-url). A gate that guards a critical section, which in turn places demand on CPU resources. CPU is a potential latent or looming bottleneck, when considering increasing the capacity of the gating resource: beyond a point, increasing the capacity of the gate (as a solitary action) may reap no benefit.
the capacity of a gating resource, we also simulate Add Load actions in that fashion.

There is a second way to reduce contention for a gating resource: tuning the critical section. Any thread that has “passed through the gate” will simultaneously place demand on multiple resources of a single machine. For example, while a monitor owner executes a CPU intensive critical section, it places demand on both the monitor and CPU resources, as illustrated in Figure 5. We represent this simultaneous ownership with a special marble, called a ghost.

A ghost marble is a normal marble but with the additional property of being related to exactly one other marble $m$; $m$ is either mapped to a multiplexing resource, or, recursively, to another ghost marble, e.g. in the case of nested locks. We refer to this as the critical section relation.

For monitors, this information is represented explicitly in Java thread stack snapshots: the list of live monitors and, for each, the owning and waiting threads. For other gating resources, such as thread and connection pools, this information is only implicit and must be inferred (see Section 8.2).

![Figure 5. With gating resources, such as monitors and thread pools, one thread can simultaneously place demand on multiple resources: a share of the gate itself and the resources necessary to service the critical section guarded by the gate. We create ghost marbles to represent ownership of the gate resources. Those threads waiting to pass through the gate are placed in an associated queue resource.](image)

### The Critical Section Tuning Effect

Let $m$ be a marble that is benefited by a service time reduction according to The Sink Resource Speedup Rule. Thus this marble will be split in two, and $a = 1 - \frac{1}{P}$ will flow to Sink.

If $m$ lies in the range of the critical section relation, then also flow marbles according to The Demand Shifting Effect, using this factor $a$.

### 7.3 Divergent Demand: Inflow Exceeds Outflow

The third and final scenario of excess demand affects only server workloads. If new work arrives more quickly than existing work can be dispatched by the processors, work will begin to pile up. This pile-up will only reach catastrophic proportions if the long-term average product of arrival rate and service time exceeds the capacity of any resource in the system (an application of Little’s Law): response time will diverge to infinity.\(^7\)

A thread pool bounds this badness, providing a way to specify the maximum factor by which a resource should be overcommitted. In this case, for a pool of capacity $P$ serving as a gate, through which requests must pass, over a resource with capacity $C$, the factor is $\frac{P}{C}$; in Table 1, this overcommit ratio is 5, meaning that we are willing to overcommit our CPUs by a factor of 5. The resulting amount of Demand that will be witnessed is the minimum of $P$ and product arrival rate and service time.

Thus, as shown in the last two rows of Table 1, in a regime of divergence, demand for resources is likely to be an imperfect reflection of the underlying load. In practice, thus, we cannot parametrize tuning actions with any hope of accuracy, but this does not render the actions themselves moot. For example, diverging due to CPU saturation is a good sign that this is where tuning effort should be focused, even if we cannot predict how many processors to add. This topic, of tuning plan generation, is the topic of Section 9.

### 7.4 How Accurate are Our Predictions?

We now demonstrate how accurately our implementation predicts speedup and residual demand. For these experiments, we use a microbenchmark that simulates three concurrent Work Units executing in either a server or batch workload. In one experiment, the Work Units exercise only multiplexing resources, and in another we throw a gating resource, in the form of a lock, into the mix.

The experiment works as follows: we selectively eliminate certain of the Work Units; e.g. what would happen if the first and third Work Units were eliminated? We reference these tuning plans in a binary notation: 010 denotes that particular what-if scenario. There are thus six plans for each type of workload (2\(^3\) excluding 111 and 000, i.e. no tuning, and nothing to do). We measure the actual speedup in each case (relative to 111), and also measure the actual residual demand (by observing the call stack samples after the tuning plan has been implemented). We compare these realities to what our simulation says about speedup and demand.

Figure 6 summarizes the accuracy of our predictions of speedup. Observe the close tracking of actual and predicted speedup: the linear regression correlation coefficient between the two is 0.96.\(^8\) For the most part, the predictions

---

\(^7\) This divergence due to excess load is nicknamed the “hockey stick” effect.

\(^8\) In this experiment, underlying each data point in the chart is at least 5 runs, each with 7 snapshots, and each of those contain 36 call stack samples.
underestimate speedup. Figure 7 continues this experiment, showing good predictions of residual demand.

Knowing the nature of the workload is important for accurate predictions. As shown in Figure 8, if we mistakenly treat a batch workload as if it were a server workload, our predictions of residual demand are poor.

8. Infering Work Units and Demand

As Section 6 and Section 7 showed, to simulate the benefit of a plan, we require a collection of traits to be associated with the input data:

- **Work Unit** of a call stack: From the standpoint of increasing performance, we primarily care about those threads observed to be servicing a work item of the application. It is these units of work that we desire to make faster, or, as load increases, the tasks for which we need to maintain acceptable levels of performance.

- **Resources Demanded** by a call stack: The speedup and residual demand of a tuning action depend upon the level of concurrent demand placed on each software and hardware resource.

- **Workload Type** of the application as a whole: Section 6.1 and Section 6.2 showed that the effect of a tuning action depends upon the workload: server or batch.

The simulation aspect of the Marbles Planner is not hardwired to depend on any particular source for this information. However, the input data we use does not represent these traits explicitly. Thus, some inference work is necessary to pull this information out of the call stacks.

Even traits as seemingly simple to know, such as whether the thread is Runnable (in the sense that the thread is either executing on CPU, or is on the run queue) are not immediately obvious from a thread stack sample. Often, this particular aspect of demand is nominally available, but it is usually unreliable. The JVM has to quiesce the threads be-
For writing out a consistent snapshot of the call stacks of live threads. Knowing both the run state of a thread and its call stack, as a consistent pair of information, is challenging. For this reason, many JVMs falsely report Runnable threads as being not-Runnable, and vice versa; earlier work first addressed this particular issue.

To infer Work Unit and Demand from a call stack sample, we use a rules-based approach. Each rule codifies a mapping from an invocation (the triple of package, class, and method name) to a Work Unit or demanded resources. This approach has proven effective, without an explosion in rules, because most Java applications adopt common class naming conventions, and use a common set of frameworks.

A rules-based approach also permits easy tailoring to handle the strange coding practices that crop up in applications from time to time. For example, when analyzing our own code for performance issues, we realized that we used fairly non-standard naming conventions for our tasks. Adding a half dozen Work Unit rules quickly resolved this issue.

We will now describe the rules systems for inferring Work Unit and Demand, and show how, in each case, to know when the rules are likely to be insufficient.

### 8.1 Inferring Work Units

We provide a built-in set of rules that map invocations on a stack to the Work Unit being serviced by that stack sample. The rule engine scans a stack, from leaf to root, and applies the rules shown in Table 2. Each rule matches on the package, class, and method names of the method invocation of a call stack frame. When a match succeeds, the rule asserts a Work Unit label for the stack as a whole.

For example, if a stack contains an invocation of LoginServlet.doGet, it would be reasonable to conclude that this sampled thread is servicing the Login Work Unit. The third row in this table covers the classes for servlets that are generated from JSP (Java Server Pages) source by two common servlet containers (IBM WebSphere and Apache Tomcat).

<table>
<thead>
<tr>
<th>Frame Matching Rule</th>
<th>Work Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class.contains(Filter</td>
<td>Handler</td>
</tr>
<tr>
<td>Class.Method.equals(doGet</td>
<td>doPost)</td>
</tr>
<tr>
<td>Package.startsWith(com/ibm/jsp</td>
<td>org/apache/jsp)</td>
</tr>
</tbody>
</table>

| Table 2. The built-in Work Unit rules match on the package, class, and method name of the method invoked in a frame. When a rule fires, it assigns a Work Unit label to the stack. |

It is possible for one call stack to have more than one matching frame. Multiple matches occur for two primary reasons: data-driven dispatching, and filters. The former occurs when, for example, a single servlet acts as a dispatcher to nested work units. Rather than having a separate servlet for each work unit, this coding architecture creates a single servlet that in turn hands control to the actual Work Unit, based on the parameters to the servlet invocation. A Java servlet *filter* is a stage in a pipeline of processing elements that contribute to the servicing of a main request [9]. In this coding architecture, there are often apparently a great many false positives and false negatives.

Figure 9. The number of Work Units per data set, across our suite of 15,000 data sets.

Figure 10. Sanity checking the inference of Work Unit. Very shallow call stacks that have been assigned a Work Unit are likely to be false positives. Deep call stacks that have not been are likely to be false negatives.

Unfortunately, space limitations do not allow discussion of Workload Type inference. Elevator Summary: most batch workloads are executed in phases, and a program has phases if samples are likely to have only one Work Unit.
8.2 Inferring Resource Demand

To infer the resources demanded by a given call stack sample, we use a system of implications that builds up facts about the activities being performed by the stack. We start from a few simple observations about how the thread is interacting with the outside world, and then combine these into more meaningful predicates.

For example, a thread is likely to be waiting for data to return from a remote network operation if: it is performing a read from a network socket, and it is in the middle of servicing a request, and it is executing data access logic, and it is not waiting on a monitor. The first alone at first sight seems sufficient; e.g. if we saw a socketRead at the top of the stack, it seems reasonable to interpret this literally, i.e. as a read of data from a socket, and thus conclude that the thread is fetching data.

Unfortunately, the first implicant is a necessary but insufficient predictor of demand. For example, it is common to implement a client-server pattern by having the server perform a blocking socket read. This has the effect of the server appearing to be waiting for data from a query, when it is simply waiting for work. This is an important distinction, because the former places demand on network resources (and possibly a remote database machine), and the latter places demand on no resources.\(^\text{10}\)

8.2.1 Native Implicants: The Java-Native Periphery

Table 3 presents the ways a Java thread can interoperate with the world outside of Java. This list of native operations, and the implicating methods for each, have both proven to be infrequently changing and small. Currently, the total number of rules is about 450. We will shortly show that these suffice to define the native periphery for our 15,000 submissions.

Some basic implicants are not represented by an invocation, but rather in metadata associated with a call stack sample. In particular, how a thread interacts with monitors is not entirely a function of which methods it has invoked; e.g. one thread waiting to enter a synchronized lexical scope, and a second that is executing in that critical section — both of these will have the same method on the top of their stacks.

8.2.2 Data Access: isDataAccess and Beans

In our experience, the most difficult distinction to draw is that of a thread waiting for data to return from a remote data query versus a thread waiting for work to arrive. Usually, accessing data requires going through some sort of data access layer. These layers translate application requests to accessing data requires going through some sort of data access layer. These layers translate application requests to

\(^{10}\)Modeling memory consumption is a worthy subject of future research.
### Table 4.

We establish levels of demand by directly observing or inferring demand predicates, such as isRunnable and isWaitingForData. In addition, for gating resources, we need to distinguish between consumption and desire for consumption.

<table>
<thead>
<tr>
<th>Demand</th>
<th>Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>isRunnable</td>
</tr>
<tr>
<td>Disk</td>
<td>isWaitingForLocalData</td>
</tr>
<tr>
<td>Network</td>
<td>isWaitingForRemoteData</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Demand</th>
<th>Predicate</th>
<th>Implication</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock</td>
<td>isWaitingToEnter</td>
<td>Queued up</td>
</tr>
<tr>
<td></td>
<td>isOwner</td>
<td>Share Consumed</td>
</tr>
<tr>
<td>Thread Pool</td>
<td>isWaitingForWork</td>
<td>Share Available</td>
</tr>
<tr>
<td></td>
<td>isInRequest</td>
<td>Share Consumed</td>
</tr>
<tr>
<td>Connection Pool</td>
<td>isWaitingForConnection</td>
<td>Queued up</td>
</tr>
<tr>
<td></td>
<td>isWaitingForRemoteData</td>
<td>Share Consumed</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Implied Predicate</th>
</tr>
</thead>
<tbody>
<tr>
<td>isNetAccept &amp; !isInRequest</td>
<td>isWaitingForWork</td>
</tr>
<tr>
<td>isNetRead &amp; !isDataAccess &amp; !isInRequest</td>
<td>isWaitingForWork</td>
</tr>
<tr>
<td>isNetRead &amp; isDataAccess &amp; isInRequest</td>
<td>isWaitingForData</td>
</tr>
<tr>
<td>!isWaitingForData &amp; !isWaitingForWork &amp; !isWaitingToEnter &amp; ...</td>
<td>isRunnable</td>
</tr>
</tbody>
</table>

8.2.3 Demand Rules

From these basic implicants, we can begin to construct the rules that predict the resources demanded by a call stack. Table 4 summarizes the multiplexing and gating resources considered in this paper: CPU, Locks, Disk, Network Requests, etc. We associate a predicate with each aspect of demand; e.g. a stack with the isRunnable predicate will be assumed to place demand on the CPU resource. Table 4(c) gives some example implication rules that we use to infer these predicates of demand.

For all resources, we need to infer the level of demand. For gating resources, we need an extra property, the amount of pent-up demand. Section 7.2 shows why this information is necessary to faithfully simulate the effect of a tuning action upon a gating resource: it is the pent-up demand that shifts to the resources of the critical section. Table 4(b) summarizes our approach for distinguishing consuming demand from pent-up demand.

For locks, the level of pent-up demand is given to us directly in the stack sample data: the threads waiting to enter a monitor are indicated as such. For thread pools, we can know the number of idle worker threads available by simply counting the number of thread samples with the isWaitingForWork implicant.

8.2.4 Gauging the Accuracy of Demand Predictions

To gauge the success of this approach for inferring the demand relation, we need to measure how accurately it predicts actual resource consumption. Figure 11 shows a histogram of the disparity between observed and inferred level of CPU demand, limited to the verifiable reports — i.e. those for which we have true CPU utilization information.

![Figure 11. The disparity between observed and inferred level of CPU demand, limited to the verifiable reports — i.e. those for which we have true CPU utilization information.](image-url)
of our data sets, arranged into buckets according to the disparity between actual processor utilization and the level of predicted by the mapping rules. We gauge actual processor utilization by observing the output of tools such as `vmstat` on UNIX, or `typeperf` on Windows. For this reason, we can only validate against the 6000 (of our 15,000 reports) that have this CPU utilization information.

For the large majority of the submissions, 5362 of 6077 (88% of them) the inferred consumption relation predicts actual consumption within 20%. For 3499 of the submissions (58% of them), the inferred value differs from actual level of processor utilization by at most 5%.

Figure 12 shows a breakdown of the demanded resources, across our data sets. There is always a plurality of resources consumed. Note that it is rare for applications to suffer from over-consumption of a single resource. Rather, is the norm for an application to consume a wide variety of resources. These consumed resources include those internal to the process, such as locks, local to the machine, such as CPUs and disks, and those external to the machine.

9. Automatically Generating Tuning Plans

The discussion of tuning action simulation in Section 6 and Section 7 assumed that a plan was provided as input to the simulation. This works well for analyzing what-if scenarios that hypothesize future possibilities, but isn’t as helpful when trying to resolve a problem in the here and now. In this section, we present an algorithm for automatically generating tuning plans whose aim is to eliminate extant bottlenecks. The plan generation algorithm strives to maximize benefit, while heuristically minimizing the amount of needless expenditure of tuning effort.

9.1 Parametrizing Single-action Tuning Plans

When generating a single-action tuning plan, we need to enumerate the possible actions, and, for each, choose a value for its tuning factor. A good choice of factor is one that isn’t “overkill”. A tuning action is overkill if it buys no additional benefit over lesser plans.

Overkill and The Goldilocks Factor

Let $f$ be the tuning factor of a given tuning action. We consider this tuning plan to be overkill if there exists a tuning factor $f' < f$ such that the Speedup (as considered by The Sink Resource Speedup Rule of Section 6) when using $f'$ is at least that when using tuning factor $f$. Whereas, if there does not exist an $f'$ whose Speedup exceeds that of $f$, then $f$ is the maximum effective tuning factor — it is just right.

Using this somewhat blunt definition, the maximum effective tuning factor of a capacity action is exactly the overcommit factor of the resource: if there is twice as much concurrent demand for CPU as there is available capacity, then we needn’t increase the capacity of CPUs by any more than this factor of 2.

For tuning actions that directly reduce service time, the maximum effective tuning factor is unbounded: e.g. the faster the code becomes, the more effective the action is at reducing service time. With a more sophisticated specification of desired performance goals, this can be refined. For example, if the user states that Work Unit W needs a 2x decrease in service time, we can establish a maximum effective tuning factor for service time-reducing actions.

9.2 Tuning Actions Specificity

Next, to enumerate the set of possible action types, we inspect the mapping from call stacks to resources demanded. The range of this mapping defines the set of resources in the system. We thus evaluate one capacity action to each resource, whether multiplexing or gating, parametrized with the maximum effective tuning factor at that resource.

To select service time-reducing actions, we have two choices for the specificity of the action. We can either render the resource itself faster, or we can render the operations faster. We employ the following heuristics. First, we consider it unlikely that resources can be made magically faster. When simulating what-if scenarios, this may be a valid action, and is not precluded in the simulation. However, from the standpoint of suggesting tuning plans for the “here and now”, we have chosen to avoid including actions that speed up resources.

We are thus left with actions that increase the speed of code. But which code? Here, we employ a second heuristic. We consider either code that is demanding CPU or code that is performing a data access (according to subsubsection 8.2.2). For the former, we create one tuning action for every leaf-most method; i.e. for every operation that is executing on the CPU, we consider a tuning action that renders this specific operation faster. For the latter, the data accesses, we create one for every Bean; i.e. for every data type being

\[11\] We believe it is easy to generalize this discussion to replace the notion of strict overkill with one of marginal added benefit. Doing so would require some guidance from the user, to shape these cost-benefit trade-offs.
reconstituted via a remote data access, we consider a tuning action that renders this specific query faster.

9.3 Creating Multi-action Tuning Plans

We now have a set of parametrized, single-action, tuning plans. We evaluate the Speedup of each, via the simulation of Section 6 and Section 7. It is possible that the Speedup will be zero. For every zero-benefit tuning plan, we consider whether there was a latent or looming bottleneck.

Latent and looming bottlenecks can only occur for actions that result in demand shifting from one resource to another. To witness of a latent bottleneck: observe a tuning action that results in some demand shift but has a Speedup of zero. In these cases, we must add at least one more tuning action to the plan, in order to maximize the effectiveness of that first tuning action.

For this paper, we consider the remedy for a latent or looming bottleneck to be the capacity actions, focused on those resources that previously were not overcommitted and now are. Each of these added capacity actions is parametrized according to the maximum effective factor.

We repeat this process iteratively until the final action incurs no latent or looming bottlenecks.

10. Using the Marbles Capacity Planner

Finally, we discuss two main uses of the on-the-fly capacity planner presented in this paper. The first is as a filter over problems: by generating candidate plans to fix performance problems, we can also know which call stack samples will not benefit from any plan. The second use that we discuss is as the foundation for exploring the impact of constraints on the planning engine: every what-if scenario takes the form of a number of bounds on the plan generator.

10.1 Plan Generator as Filter: Has-No-Benefit

The plan generator has an important use as a filter over potential problems. If there does not exist a plan, of any size or action constituency, that benefits a stack, there is little point in displaying this stack to the user.

For example, a common way to implement the pattern of waiting for work involves lock contention. A worker thread, when it becomes available, attempts to acquire a monitor that guards the work queue. In this situation, fixing the lock contention, even in tandem with other actions, will neither have benefit on the response time of any Work Unit, nor will it benefit the throughput of the system. The Marbles Planner infers this, as it futilely attempts to generate plans addressing this lock contention. We refer to this property as hasNoBenefit.

Figure 13 summarizes the fraction of call stack samples in a given data set that the Marbles Planner estimates as having the hasNoBenefit property. The horizontal axis of this chart represents the CPU overcommit factor. As the factor by which CPU is overcommitted increases, there is a moderate increase in the fraction of samples that benefit from tuning. Also observe that, across the spectrum of overcommit factors, a majority of samples in every data set are not worth tuning.

Thus, when presenting findings to a user, even if the user does not care about the plans themselves, she is gaining benefit from the planning system. She no longer has to be concerned with misleading findings in the tool. It is often the case that this kind of “mock” lock contention is prevalent enough so as to dwarf the real problems.

10.2 Exploring What If Scenarios

We now demonstrate two final points. Firstly, the Marbles Planner has the capability to answer a variety of what-if questions. Secondly, asking these questions is important, because the answers, as to how to go about tuning your application, vary radically depending on the nature of the questions being posed.

Every what-if scenario places constraints on the plan generator. At one end of the spectrum, the plan generator can be given free reign to propose tuning plans with high financial cost, and in producing a high volume of proposed plans. Realities usually impose constraints on one or the other of these variables. In the next two subsections, we explore these two dimensions of constraints: the economics of affordability, and the issues of ranking extant problems.

10.2.1 What if There are Budget Limitations?

To experiment with the economic side of planning, we explore three what-if scenarios. In the first, we pose the what-if scenario where code or query tuning actions can infinitely speed up the code or query, and where there is an infinite budget for adding hardware capacity. This corresponds to no reality, but we feel that it reflects the perspective encouraged by a typical hotspot-finding tool: show me what is hot, and prioritize problems in that way. At the other extreme, we
The study then proceeds as follows. For each call stack sample, the plan generator may produce multiple tuning options; e.g. adding CPUs or tuning the CPU-consuming code; or fixing a lock or tuning the critical section. In both cases, it is likely that one plan will produce a larger benefit.

Thus, we inspect each call stack sample and place the best plan into one of five buckets: one-action tuning plans that tune a CPU-demanding operation, that tune a data access query, that fix a lock, that add capacity, or multi-action tuning plans consisting of a mixture of actions.

Figure 14(a) shows the population distribution of these buckets for the case of infinite tuning. Under the optimistic scenario of unlimited budget, the system predicts that code tuning is most often the best plan. Furthermore, as the overcommit factor of CPU increases, code tuning is even more likely to be the best course of action. This is as expected, for when CPU is saturated (corresponding to an overcommit factor of greater than 1), code tuning has a double benefit: it

Figure 15. A breakdown of the multi-action plans from Figure 14 by the number of actions in a plan.
both decreases service time directly, and reduces the penalty of multiplexing.

At the more austere end of the spectrum of economics, Figure 14(c) shows a very different story. When the overcommit factor is below 1 (meaning that CPU has excess capacity), single-action plans that focus on the code (either code tuning or fixing locks) show a peak — they are more often to compose the best plan when CPU is not overcommitted. When the overcommit factor CPU rises above 1, capacity actions become increasingly important. This is as expected: once the CPU is overcommitted, latent bottlenecks cap the benefit of single-action tuning plans. Also observe that the relative importance of single-action plans that fix lock contention problems diminishes as the latent and looming bottlenecks begin to rear their heads, once the CPU overcommit factor rises above 1.

Figure 14(b) shows the breakdown of best plans for a case somewhat in the middle. As expected, the shape of the curves shows a point on the spectrum from the austere to the unlimited cases.

Figure 15 further breaks down the multi-action plans from Figure 14. When tuning has less power to optimize (as in Figure 15(c)), three-action plans dominate. When given moderate power to optimize (as in Figure 15(b)), there is a plurality of the best multi-action plans, split between those requiring two tuning actions and those requiring three; in this case, the arithmetic mean fraction of call stacks whose best tuning plan involves two actions, across all CPU overcommit factors, is 40%; for 3- and 4-action plans, this mean is 46% and 6.1%, respectively.

10.2.2 What if Only One Plan is Allowed?

A second aspect of budgeting is the time spent investigating potential problems. All three scenarios from the previous study assume that the users could devote the time to study a separate plan for every call stack sample. This luxury shares its unrealism with the scenario that allows for infinite tuning and an unbounded budget for hardware — they are useful limit studies, but not something to be employed in practice.

Just as there is a spectrum of what-if scenarios along the dimension of cost (of which we explored three points in the previous study), there is also a spectrum along the dimension of the volume of suggested plans. At one extreme, the planner can be used, as with the previous study, to generate one plan per call stack sample. At the other end of this dimension of volume, the planner would suggest a single plan, one that is estimated to have the highest impact on performance. We explore this scenario now.¹³

---

¹³There are other interesting points along this spectrum of volume worthy of further study. For example, the planner could also be tasked with identifying a separate best plan for each Work Unit.
To identify a single best plan requires establishing a ranking order over plans, so that we may present the top-ranked plan to the user.\(^{14}\) A ranker should take into account the level of dissatisfaction that the users have with the Work Units in their application; from this, a single plan can be identified that has the highest benefit to the Work Unit with greatest dissatisfaction.\(^{15}\) For this paper, we unfortunately cannot leverage such information, as it is not represented in our corpus of 15,000 data sets.

In this paper, we introduce a simple ranking function over Work Units, and leave a full treatment of this important topic to future work. We rank problems by first ranking Work Units, and then selecting the plan with highest benefit to the most prevalent Work Unit.

### The Prevalence Ranking Function

We define the **prevalence** of a Work Unit as the sum of two separate ranks: the likelihood of occurrence rank and the concurrency rank. Given two Work Units \(A\) and \(B\), the likelihood of occurrence of \(A\) is greater than that of \(B\) if \(A\) is active in more snapshots of the thread state. The concurrency of \(A\) is greater than that of \(B\) if, while active, \(A\) is on average active in more threads than the similar figure for \(B\).

By combining likelihood and average concurrency, we seek to avoid what we term the “mountains and valleys” problem of ranking. If you use only average concurrency in order to rank, as does any simple method profiler, frequently witnessed but single-threaded Work Units will be dwarfed by infrequent but highly concurrent Work Units. Figure 16 illustrates this situation.

The charts of Figure 17 mirror those of the previous study shown in Figure 14: we experiment with the same three what-if scenarios of cost, but this time we choose the highest benefit plan for the Work Unit with highest prevalence.

For the lower-cost scenarios, i.e. comparing Figure 14(c) to Figure 17(c) it is more often the case the best plan involves fixing locks or multi-action plans (and, though not shown, these multi-action plans all include an action of fixing a lock). This implies that it is common to have egregious lock contention in our corpus of reports. Contrast this with the decreased importance of query tuning.

### 11. Related Work

There are a number of existing strategies for performance tuning, performance modeling, and capacity planning.

**Performance Tuning** There are many commercial tools available that assist in finding hot spots in code [1, 23, 24, 26, 27]. These tools collect either trace or the same sort of call stack sample information that we rely upon, and present views that aid in the identification of hot methods. When based on full trace information, they have the potential for identifying finer-grained performance problems, at the expense of considerable instrumentation overhead. Because these tools focus on execution time it is difficult to use them to analyze idle time. Some tools focus exclusively on certain aspects of idle time, such as identifying lock contention [25].

**Analytical Approaches** One approach to understanding performance problems, and planning out solutions, constructs analytical models. These models take the form of parametrized formulas whose output predicts the service times of the application [4, 8, 13, 17, 22].

**Queuing Network Models** Queuing network models [14] model resources as queues with a capacity and a service time. Capacity planning tools centered around queueing networks [6, 21, 28], often require a large degree of customization. Users must construct the model’s topology, and assert the properties each resource (queued or gating, and the elements of the Kendall’s notation [20], e.g. the discipline for prioritizing work). Then follows a period of tweaking to ensure a good fit for the predictive formulae. Though quite a bit of work, this offers the potential for precise predictions.

Just as we have discussed batch and server workloads, other work has identified the vital importance of the nature of the workload to model [19].

There are alternative state-transition models that are unsuited for our problem. Petri Nets [18] are often used to model distributed processes. The notion of colored tokens were the inspiration for our colored marbles. With Petri Nets tokens transition only as a result of demand exceeding a threshold, which does not seem to match the tuning action simulation we need. In the operations research community, conceptual models such as the theory of constraints [5] offers a close analog to the computational model of queueing networks. As with Petri Nets, the theory of constraints is intended to model consumption and flow in a way that is not well suited for capturing bottlenecks in computing systems.

### 12. Conclusions

We have introduced a system for planning performance optimizations and resource provisioning. We have demonstrated that it works well, on both a suite of predictable microbenchmarks, and 15,000 data sets from the wild. Through use of this system within IBM, we have been able to track the role of our users. We have found them to be a satisfying mix of developers, testers, operations, and support teams — and even the occasional manager.

The Marbles Capacity Planner has been implemented, and the public is free to use it: [http://wait.ibm.com](http://wait.ibm.com). You
may view the report gallery (or analyze your own applications), and, from there, view the tuning plans recommended by the planner. As discussed in Section 10.1, even if you do not view the plans themselves, the user interface is filtered based on the planner’s evaluation of the hasNoBenefit property.

The design of the UI you will see is an important topic of future discussion: we feel strongly that any planning system needs a good way to communicate its findings. Among the important topics is plan ranking, introduced in subsection 10.2.2: how can we guide the user towards what is important? We have also completed support for tuning plans that cross machine boundaries (such as adding CPUs to the bank of database machines), and are studying the effect of insufficient information on the quality of the predictions. A complete discussion of our modeling of disk resources and connection pools have also been omitted from these discussions, and can be presented in future work.

References

[21] TeamQuest. TeamQuest Model software.
[27] Yourkit LLC. Yourkit profiler.