Undergraduate Honors Thesis

Against All Probabilities

A modeling paradigm for streaming applications that goes against common notions

Rahav Dor
Advisors: Roger D. Chamberlain and Mark A. Franklin
Dept. of Computer Science and Engineering, Washington University in St. Louis
Rahav.Dor@wustl.edu

Abstract

Hardware and software design requires the right portion of skills and mental faculties. The design of a good system is an exercise in rational thinking, engineering, and art. The design process is further complicated when we aspire to build systems that exploit parallelism or are targeted to be deployed on architecturally diverse computing devices, FPGAs or GPUs to name just a few.

The need to develop systems that can take advantage of computing devices beyond general purpose CPUs is real. There are several application domains and research efforts that will simply not be able to adequately perform or yield answers in a reasonable amount of time otherwise. Developing a mathematical model of a system is a key stepping stone to a high performance system, but often is absent from the design process due to the complexity of the model development.

In this paper we offer an easy, yet solid approach to the development of such mathematical models. This adds a little bit more weight to engineering side of the design process in the form of a quantifiable method that enables designers to reason about their systems, identify bottlenecks, and gain vital information for performance improvements.

This work was supported by NIH grant R42HG003225 (through BECS Technology, Inc.). R.D. Chamberlain is a principal in BECS Technology, Inc.

1 Introduction

Performance improvement of traditional computing devices, CPUs for example, has been slowing down and we cannot count on their improvement to accelerate the applications we execute on them. If we seek better performance for our applications we are bound to consider implementing them decomposed into multiple modules, with each module designed to make maximum use of the specific computing device it can best run on. It is also the case that as part of the life cycle of a hardware or software system it will periodically be considered for an upgrade, to improve its algorithmic behavior, or to have it use computing resources that were not available before. It is a hard task to correctly model a system's behavior when it is targeted to run on a general purpose CPU and it becomes much harder when our system is to be decoupled into modules and run on architecturally diverse computing devices.

How do we reason then about the performance of a system? How can we discover that one of the modules is a bottleneck? Having this information will allow us to design an improvement, but if we eliminate the bottleneck how do we know which module will saturate next? There may be many characteristics of the computation devices we will want to take into account. Some modules may benefit from running many threads of execution when the computation device they run on supports it, but on the other hand this specific device may require more cycles to access memory. While other modules may be suitable to run single threaded, but they might benefit from faster access to memory or a larger size memory. Many other questions arise when we consider, for example, how to use the resources of

platforms such as FPGAs. Thinking of all the functional units on an FPGA as generic, then we can ask how many of those generic cells do we want to instantiate as memory for example, and how many as logic?

Despite the potential benefits of having a mathematical model of a system behavior, we see that they are often not developed in practice, largely due to the difficulties of developing and validating the models. In this paper we offer a novel approach for developing such models. Our approach is effective, easy to develop, and useful for understanding of a system and reasoning on its improvements or limitations.

Quantitative methods for reasoning about the performance of algorithms are hardly new. Queuing theory is also a very well established discipline and is predominately used to model the performance and make predictions of queues and systems behavior, from customers waiting in post office lines to modeling computing servers and their input queues.

We take a fresh approach in making the assertion that the scientific approach can be used during the modeling process. That is, a mathematical model of a system can be built and verified empirically. Then we can often use the simplest queuing models even if we explicitly know that the system we are modeling does not exactly fit the queuing model. We claim that within reason, the simplest queuing models can be and should be used.

Our technique depends upon the ability to effectively gather empirical data on the performance of the executing application. We make extensive use of the TimeTrial performance monitor for this purpose [10].

Using the simplest queuing models has advantages. These models employ simple probability distributions and the equations governing them are readily available in the literature, which greatly simplifies the development of the model and the computations involved rendering the model very palatable for designers. We hypothesize that even when the modeled system consumes input and processes data in a manner that does not match the chosen model's arrival rate and service time probability distributions, the models can still yield useful and accurate results.

This paper demonstrates one use of our approach and presents the results of modeling Mercury BLAST. BLAST is a software system authored by NIH for the purpose of finding similarities between the DNA of different species [2, 3]. Mercury BLAST is a hardware / software hybrid authored in the High Performance Computational Biology research lab at Washington University in St. Louis [6, 8, 9].

We will demonstrate that despite the use of seemingly inappropriate queuing models the model we develop correctly predicts the system behavior. We will also show that the model can tell us when it is expected to not work, a fact that makes our assertion that the simplest queuing models should be used even more compelling.

2 BLAST

2.1 NCBI BLAST and Mercury BLAST

There are several implementations of BLAST (Basic Local Alignment Search Tool). The two of interest to us in this writing are NCBI BLAST (National Center for Biotechnology Information) and Mercury BLAST. NCBI is an organization under the wings of NIH (National Institutions of Health). Mercury BLAST was developed in Washington University's School of Engineering and Applied Science, High Performance Computational Biology lab with the objectives of accelerating NCBI's BLAST without compromising accuracy (referred to as sensitivity.) We judge Mercury's sensitivity by the percent that its results are common with NCBI's results. Mercury is performing consistently above 99.9% sensitivity. Striving for excellence we also check that while we may offer more results than NCBI BLAST the only results that Mercury is allowed to miss are not biologically meaningful.

2.2 Why does one need BLAST?

The sequencing of the molecule of life, DNA, has opened new fields of research and practice for biologists trying to explain the traits and origins of the species and for pharmacologists in their pursuit of novel drugs. The field follows a logical assertion that in different species, similar DNA molecules will have similar functions. Researchers sequence the DNA of one specie, perform experiments and observe it under various conditions, and annotate the various segments in the DNA with their function. Later on they use BLAST to search for the same genes in the DNA of another specie. Assuming that in both species the same genes are responsible for the same biological function, they draw conclusions about the second specie.

The task for BLAST is to find these similar segments when given DNA of two species as input.

The challenge of achieving this objective hides in a few places. Genomes are composed of DNA, a molecule in the shape of a double helix. Different segments of this molecule define different functions of the specie. The building blocks of a DNA molecule are four chemical compounds called bases: Adenine, Thymine, Cytosine, and Guanine. A computing scientist can abstract

such a genome by a string of characters restricted to the alphabet A, C, G, T. A typical length of such a genome is billions of bases. For example the human genome has about 3.2 billion bases, and the largest known genome as of the time of this writing is of the Amoeboid Polychaos dubium (“Amoeba" dubia) with 670 billion bases. Viruses have between only a few thousands (Bacteriophage MS2) to millions of bases (Minivirus). Plants have hundreds of millions bases and even billions such as the Liliaceae Fritillaria assyrica (from beautiful Lily family) with its 130 billion base pairs [12]. The chief challenge in finding similar segments stems from the size of the inputs that BLAST needs to process, it is impractical to compare base to base, it will simply take to long.

Another challenge is presented by the evolutionary process itself. When a specie reproduces and its DNA is replicated in the process it is often the case that minor mutations occur in some of the bases. The result can be that the mutation is a killer and the offspring will not survive (in which case we are not likely to have its DNA for sequencing simply because its host has disappeared). The mutation can be benign, or it can be a great success, probably leading to an improvement in the specie. It is the benign and the successful mistake that cause computer scientists the algorithmic challenge. In two different species, these random mutation are likely to keep the gene function similar, but the actual bases will be different to some degree. The question we need to ask is how do you make a computer algorithm identify something as having the same function when it is not built from the exact same bases (character strings)?

It is key to note that the benign and great success random mutations are exactly the interesting cases. Crucially if one specie has become more adapt due to a mutation, less susceptible to illnesses, or enjoy some other good attribute – we certainly want BLAST to discover as many of those same-function different-bases genes. They could lead to superior drug or genetic engineering that will better many people’s life.

Another computational challenge stems from the fact that biologists are interested in the longest possible strands of genes that are similar. The hypothesis is that the longer the strand is, the more likely we are to have indeed found a biologically meaningful gene function, and less likely we are that this similarity is unrelated and occurred due to chance alone. Since the real DNA is one very long, sequence of bases, with limited understanding of the markers at the beginning or end of a gene function, the algorithmic challenge to address is how long is biologically meaningful? Declare too short of a strand as a match between two species and you may have mistakenly attributed different genes with the same function. Certainly a risk if you are going to genetically intervene with that gene. Look for too long matches and you may not find any, or run the risk that more than one biological function is masked in that segment.

2.3 BLAST high level architecture

One possible design of a system that addresses these challenges is the design of NCBI BLAST. See figure 1.

The first module, stage 1 or the Seed matching module, is looking for an initial short match between the two specie’s DNA. Instead of the regular way of finding matches between two strings by checking a pair of characters (each DNA base is represented as character) and progressively increasing the matched length up to the point of the first non-matching character, BLAST looks for matches of minimum w-characters (w-bases) long. These matches, being not shorter than w characters are called seeds or are often referred to as w-mers. The desired minimum seed size, w, can be specified as a run-time parameter, with the default being 11 bases long. The heuristic suggests that even if some matches are lost because shorter matches are being missed, they are by all likelihood not going to turn out to be biologically meaningful. This is the first step in addressing one challenge, the need to finish the computation in a reasonable amount of time.

Stage 2’s job is to extend the seed found to a length that has the potential of being biologically meaningful. Each w-base long seed is extended in both directions without allowing for gaps. Not allowing for gaps means that each pair of bases will continue to be compared whether they are a match or not. The algorithm does not re-align the positions in the strings when a match is found after a mismatch. The algorithm counts the number of matches and mismatches while extending the seed. Two conditions must be true for the extended seed to pass to the next stage of BLAST. The count of matches needs to be above a certain value and the count of mismatches needs to be below another. These two values gives the extension a score. If the maximum scoring sub-string has a score higher than the user provided threshold, the seed is passed to the next stage of BLAST. The extended seeds that pass this tests are called high-scoring segment pairs (HSPs) or ungapped alignments.

Stage 3 performs gapped extension. At this point in the process BLAST enters the most algorithmically intensive step, but due to the relatively few HSPs that are left the computing resources it requires are relatively less intensive. Gapped extension means that the

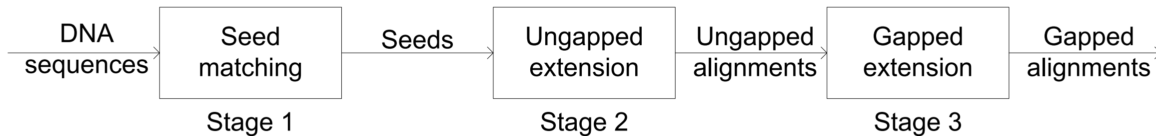

Figure 1. BLAST high level architecture

algorithm is allowed to insert and delete (called edit) bases. The number of edits are counted and HSPs that were successfully extended with a high enough count of matches and low enough count of edits are considered biologically meaningful. These segments are called gapped alignments. Longer strings with high match count and low edit count are more likely to have biologically meaningful function. This stage of the algorithm addresses the challenge explained earlier, the random behavior of the evolutionary process. So genes with presumably the same function, but with some different bases, are still found and reported to the researcher. Furthermore, the numeric score which is a direct function of how close the matches are to each other, gives the researcher a quantifiable way of assessing the similarity.

2.4 Mercury architecture

Mercury high level architecture is similar to NCBI's BLAST, but the algorithms and data structures are somewhat different. See figure 2. Mercury implements stage 1 and 2 on an FPGA (Field Programmable Gate Array) and thus exploits the inherit parallelism of hardware logic.

In Mercury, stage 1 is still responsible for finding seeds, however their length is exactly 11-bases long. Mercury uses a Bloom filter [5] to check for these exact matches. A Bloom filter can be thought of as a container of all the elements of a set. Computationally it is an array of m bits. For each element that we want to add to the set we compute k different hash functions ($k < m$). The values obtained from the hash functions correspond to positions in the Bloom array. Adding an element to the set simply turns ON the corresponding bit positions in the Bloom array. To check if an element is in the set we calculate the k hash functions and test if all of those array positions are ON. Designed for efficiency of both membership detection and memory, Bloom filters use a single array of bits to represent all the members of the set. If an element is in the set then it will surly be found because the corresponding bits in the filter are ON. But because the same array is used to store the bits of all the elements, it will also return some false positive results. That is, some queries for membership of a certain element in the set will return the answer YES, while the element is not really in the set; it is the bits turned on by other members that caused the false positive answer. Mercury utilize the inherit parallelism of hardware logic by checking all of the k bits in parallel [6, 8].

To eliminate the false positives Mercury checks all the seeds that were reported as a match by the Bloom filter against a traditional hash table. That hash lookup will return an answer that an element exists only if it really does. But hash tables pose another complication, due to the nature of hash codes, collisions could occur (collision means that two elements are mapped to the same entry in the hash table). To check if a seed is a false positive (in this case it will not be found in the hash table) or not (will be in the hash table) we check it against the primary hash table. If that position is not marked as position occupied by collisioned elements the seed is immediately forwarded to stage 2. However since a hashing function is a mapping from a large set of elements to a smaller set of codes, there are always going to be some collisions, so on average a check in the primary table will require additional memory accesses. If the hash function is well suited to the task there are going to be a small number of collisions and the average number of memory reads is going to be very close to 1.

Mercury's stage 2 task is the same as NCBI's, however the extensions do not have arbitrary long length, rather they are limited to a frame of 64-bases. Implemented in hardware the extension and score calculations can accept a new seed each clock cycle. The longest extended seed within the bounds of the frame with the highest score is selected. If its score is above a desired score threshold, then it is passed along to stage 3 [9].

Stage 3 is the original NCBI BLAST stage 3, it was hypothesized that by the time that a seed passes all the way to this stage the computation involved will not tax a regular CPU and no critical acceleration will be necessary [8]. Mercury therefore simply sends the HSPs that pass stage 2 to the original stage 3.

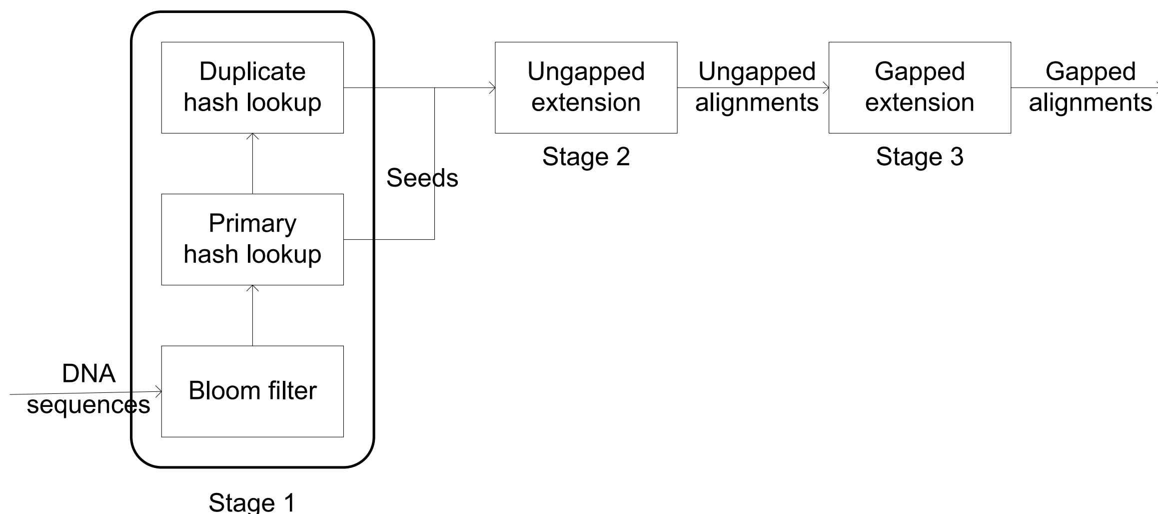

Figure 2. Mercury high level architecture

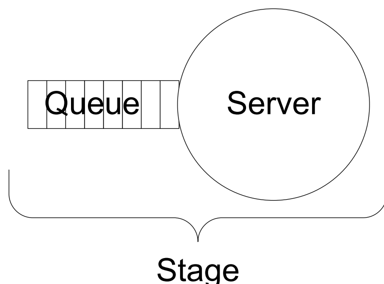

Figure 3. A stage is a queue + server

3 Queuing Theory

In queuing theory we use probability to study the behavior of waiting lines (called queues) and the service stations that attend to these queues. Queuing theory can be used to predict the behavior of familiar queues such as a line in the post office, bank, or supermarket. The results of such analysis can be used to plan the organization's service capacity, number of open service stations, manpower, and so on. Queuing theory is also used in computing systems to model the behavior of central servers, or processors, and the memory (the queue) allocated for holding data elements waiting to be served by these centralized resources [1].

3.1 The Queue + Server idea

We treat each module of a system as a server and the memory it has for the elements as a queue. The capacity includes the number of elements that can wait in the queue or are being processed by the server. The combination of the queue and the server is called a stage (see figure 3). Variables depicting values of a queue are given the subscript Q, values of the server S, and of the stage G.

A few points that are important for the correctness of later computation are in order. The stages are connected to each other by their queues, but each server is allowed to deliberately filter out some or all of the elements it process when it determines that they should not continue to the next stage. We use a preservation of flow principle which means that elements along the computation path are not lost due to any other reason (due to insufficient space in the queue for example). Furthermore, while a system may have feedback mechanisms between the stages, for example a system can assert a back-pressure signal asking a preceding stage not to send any more elements, these physical mechanisms are not modeled.

In queuing theory we use models to approximate real systems or situations. In this work we are concerned about the system when it is in its steady state. So transient behavior such as loading initial data (analogues to the clerk arranging his desk in the morning before starting to service his or her customers) is not taken into account. Stability requires that the system is not over its saturation point, in other words the data input rate is smaller than the processing rate rate. Mathematically if the input rate is λ and the processing rate is μ, then steady state means that $\lambda < \mu$ or $\frac{\lambda}{\mu} < 1$.

3.2 Queuing models

There are a number of queuing models depending on the characteristics of the stage being modeled. To the

queuing theory practitioner the following characteristics of the system determine which queuing model is to be used. Typically the model being used is denoted with the notation $A/B/S/K/N/D$ [1].

A denotes distribution of the Arrival time gaps between elements (often called inter-arrival time). To determine the arrival distribution we ask question such as: Is the time difference between input elements deterministic, with a fixed time gap between elements? Or do they arrive in a random way such as a Poisson process?

B, the service time distribution, is determined by the same questions as we asked about the arrival distribution, only this time our focus is on the service time. It is the nature of how long does it take to handle a single element by the server, and how is this service time distributed.

S denotes the number of Servers available to attend to each queue.

K denotes the max number of elements that can wait in the queue or be in the server as they are being processed. This parameter is also called the system capacity.

N is the number of elements that are available as input to the queue. For example, how many people are available to visit the post office during a certain operation window. This parameter can have a finite value or be infinite.

D is the service Discipline (e.g. FIFO, LIFO, Random). It signifies how the next element to be processed is selected.

To symbolize the model being used one would specify the characteristics of the system using the notation shown above, but shorthand is customary. For example M/M/1 denotes Poisson inter-arrival probability distribution, exponential service time probability distribution, 1 server, and room for infinite number of elements in the stage (note that this last parameter to the model is often dropped if it is clear from the context so M/M/1 and M/M/1/∞ are interchangeable). M stand for Poisson because the Poisson distribution is also know as Markovian in the sense that it does not have memory. The appearance of an element is random in time, and the appearance of the next element is independent (hence memory-less) of the time the previous element appeared. The same argument applies for the server's service time.

3.3 Solving a queuing model

There are well known equations that are used to solve the various queuing models. Some are more complicated algebraically than others. In general if we know, or can measure, enough of the variables, then we can predict an equal amount of unknowns. Furthermore, we can compute additional variables by their relationships to ones we already solved for. As an example if we know the mean arrival rate to the queue and the mean service rate of the service station, then we can compute the server utilization. From the server utilization we can predict, for example, the probability that there will be n elements waiting in the queue at any given time, compute the expected wait time of an element in the queue, and so on.

We are interested in modeling a system that is decoupled into modules and data elements pass from one stage to another (hence the name streaming applications). Such system topologies can be built as a network of queuing models with the queues connecting the output of one stage to the next [4]. We interchangeably use either the word *stage* or *module*, as appropriate in the context, to refer to any one of these physical system's modules. Each stage has its own server and a queue (of potentially length 0).

In figures 4 and 5 we present the equations governing the queuing models M/M/1 and M/M/1/K adapted from [1].

3.3.1 M/M/1

Figure 4 displays the equations needed to solve an M/M/1 queuing model (Poisson arrivals, exponential service time, 1 server attends to the queue). This model assumes that the queue length is infinite (that is, the stage has room for infinite number of elements).

When the physical module this stage is modeling has a finite queue capacity we model the probability that a back-pressure signal is asserted as equal to the probability that there are K or more data elements in the stage, where K is the total number of elements that the physical module has room for in its queue and server memory. It is not typical to use K, a limiting factor on the capacity of a stage, in an infinite queuing model. But it is an extension that has very useful results. There are at least two immediate benefits stemming from the facts that physical systems do have limited capacity and the model does not. Since the model is not bounded it allows us to correctly calculate the probability that a back-pressure will be asserted. Since the system is bounded, a difference between the modeled number of elements in a queue and the measured value indicates that the model is beyond its predictive range.

3.3.2 M/M/1/K

Figure 5 displays the equations used to solve a M/M/1/K queuing model (Poisson arrival process, exponential service time, 1 server attends to the queue, and space in the stage (queue + server memory) for a maximum of K elements).

We use the actual input rate to a stage, λ_a, as a given and calculate the offered arrival rate, λ_o. This is consistent with previous approaches [7, 11]. It also sits well with the TimeTrial [10] system which measures the actual input rate and thus we gain a safe comparative mechanism.

The maximum stage capacity, K, is the sum of the data elements that can be held in the queue plus the server memories.

In queuing theory the maximum number of elements that a stage can hold, including those elements waiting in the queue or being already processed by the server is K. Any element arriving when there is no more room in the stage is lost. In Mercury however, and in many other real computing systems, a stage will assert a full signal toward preceding stages signaling them ahead of time not to send additional elements. It is because of this behavior that we can treat P_{BP} as the probability that a back-pressure signal is asserted when we use the M/M/1 model. This calculation is not readily available in the M/M/1/K model. A counter-intuitive result given the fact that a M/M/1 model does not have the notion of a certain queue size, K, and M/M/1/K does. It is worth noting that the probability that an arriving data element is lost, P_K, is different from the probability of a back-pressure, P_{BP}. The first is the probability that a certain value, K, will occur while the latter is a cumulative probability. That is, the first is a density function while the latter a cumulative density function.

4 Our approach

The hypothesis we are trying to justify is that simple M/M/1/K or M/M/1 queuing models can be used to model systems when a precise model is not necessarily needed. We hypothesized that even in cases where a physical system's module is known to not match these queuing models, it might correctly predict the system's behavior. We want to show that used in conjunction with the scientific method we will be able to use these easy-to-solve queuing models to analytically describe the system behavior. When we say follow the scientific method we mean the process of modeling a system based on observations, verifying the model correctness by empirical results, then using it to make some prediction about the world. In our case the world is either the system as it is at this point, in which case we model its performance and identify bottlenecks. In future work we plan to use the method presented here to explore design alternatives and suggest improvements to the system based on result from the mathematical model.

Like many real-life systems the system we choose to model in this paper is very complicated. In each stage there are many algorithmic steps that we are going to abstract away. We make our work further risky by asserting that not all known aspects of the system or algorithm should be taken into account for the model to be useful. Taking the dominating behavior of a particular module will be the key to for the model to produce correct predictions. Such abstractions are already present in the earlier specifications we articulated for Mercury. The way we transfer these abstractions to mathematical expressions will become clear later in this work when we present the method and the level of details we used to model our example system. These abstractions are useful because they allow the designer to focus on the important features of her or his system. They also allow us to make our approach even more palatable because it keeps the mathematical manipulation simple. As long as these abstractions, or simplifications, are done within reason they will still allow the model to provide excellent predictions.

We selected to use the queuing models titled M/M/1 and the little bit more complicated one M/M/1/K because they, as well, are simple to solve. These queuing models a-priori do not depict the characteristics of Mercury. For example, the arrival distribution to stage 1 of Mercury is known to be deterministic, it is a constant stream of data arriving from disk. The service time of stage 1a and 2 are deterministic as well. It is still unknown what is the service distribution of stage 1b, but it is clearly data dependent as will be shown later and so may take different distributions for different runs.

Another form of simplification we advocate is that the system's physical modules do not necessarily need to have one-to-one mapping to queuing stages. In this paper for example we started with a queuing model of four stages. We later altered it by adding an additional stage and in doing so gained more insight into the inner working of one of the system's modules. Of course with less stages comes less visibility into potential bottlenecks and room for improvements, so adding or removing a stage is a balancing act between how much one would like to know about the physical system and the complexity of the model.

To calibrate the queuing model we use TimeTrial to measure the input arrival rate λ_{in} at the beginning of the pipeline, operating values of stage 1b that

Mean arrival rate	λ	
Mean service rate	μ	
Stage utilization	$\rho = \frac{\lambda}{\mu}$	
Maximum stage capacity	K	
P[n elements are in the stage]	$P[N = n] = (1 - \rho)\rho^n$	$n = 0, 1, 2, ...$
P[n or more elements are in the stage]	$P[N \geq n] = \rho^n$	$n = 0, 1, 2, ...$
P[back-pressure asserted]	$P_{BP} = P[N \geq K]$	
Mean #elements in the staGe	$N_G = \frac{\rho}{1 - \rho}$	
Mean #elements in the Queue	$N_G = \frac{\rho^2}{1 - \rho}$	

Figure 4. Set of equations for M/M/1

Actual mean arrival rate	λ_a	
Offered mean arrival rate	$\lambda_o = \frac{\lambda_a}{1 - P_K}$	
Mean service rate	μ	
Actual stage utilization	$\rho_a = \frac{\lambda_a}{\mu}$	
Offered stage utilization	$\rho_o = \frac{\rho_a}{1 - P_K}$	
Maximum stage capacity	K	
P[n elements are in the stage]	$P[N = n] = \frac{(1 - \rho_o)\rho_o^n}{1 - \rho_o^{K+1}}$	$n = 0, 1, 2, ..., K$
P[arriving data element is lost]	$P_K = P[N = K]$	
Mean #elements in the staGe	$N_G = \frac{\rho_o[1 - (K+1)\rho_o^K + K\rho_o^{K+1}]}{(1 - \rho_o)(1 - \rho_o^{K+1})}$	
Mean #elements in the Queue	$N_Q = N_G - (1 - P[N = 0])$	

Figure 5. Set of equations for M/M/1/K

are data dependent (average number of lookups in SRAM per seed $\frac{SRAM_{rd,active}}{1b_{in,active}}$, and the SRAM utilization $SRAM_{rd,median\%}$ or $\frac{1}{SRAM_{rd,median\%}}$ if we would like to use it as a multiplier), and the branching probabilities ($P_{\bar{f},1a}$, $P_{\bar{f},1b}$, and $P_{\bar{f},2}$). For example see figure 6 and 11. In the tables presenting the inputs to the module such as figure 11 we already display the stages service rates resulting from using the data dependent parameters in the calculation.

To validate the model we compare the model predictions to the empirically measured input rates ($\lambda_{a,1a}$, $\lambda_{a,1b}$, $\lambda_{a,2}$) and server utilizations ($\rho_{a,1a}$, $\rho_{a,1b}$, and $\rho_{a,2}$) of stages 1a, 1b, and 2 and queue occupancies of stages 1b and 2 ($N_{Q,1b}$ and $N_{Q,2}$).

Each of the above measurements is made for two distinct test cases. The runs are as follows:

- Run 1: The first dataset is the human chromosome 1 (from build 19 of the human genome) divided into 7,964 65,400-base segments as the query. The database consists of the 9th build of the mouse genome (2.7 GBases).
- Run 2: The second dataset consists of comparing all the non-mammal vertebrate mRNA split into 8,608 65,400-base segments as the query. The queries were searched against all the mammal mRNA in the NCBI RefSeq repository (791 Mbases) as the database.

4.1 Queuing Model of Mercury

We start by modeling the data as it arrives from disk through the PCI-X bus. See figure 6. It was important for us to model the PCI stage to be able to predict whether Mercury is working (or not) close to an optimal state corresponding to varied input rate from the delivery bus.

In all the models we will present in this work we represented a physical system module as a queuing stage if the module has some distinct properties. Properties that we might want to consider improving upon (performance wise for example) or to consider it for implementation on other computing devices (such as a GPU instead of the FPGA for example). This approach allows the designer to play with each stage's properties, examine alternative designs, and explore possible improvements for each stage separately. It also can clearly demonstrate which stage is saturating first, what could be its limiting factors (that is, probable reasons for its saturation), and so on. As will be explained later this also allows the designer to force any desired input volume into each of these stages, thus examining their behavior under various input rates.

Referring to figure 6, in this model the physical stage 1 of Mercury was decoupled to two queuing stages, $1a$ modeling the Bloom filter, and $1b$ modeling the hash lookup. Physical stage 2 is mapped to a single queuing stage. Stage 3 was not modeled, but it appears in the figure to denote the flow of elements toward it from stage 2 and to alert the designer to the fact that the PCI stage is a shared resource that both feeds the input to Mercury, as well as sends Mercury's output to NCBI BLAST. It is exactly those situations of a shared resource that the model needs to take into account. The PCI bus presents a really interesting scenario because the larger the input volume to Mercury is, the larger the potential that stage 2's output will require more of the PCI bandwidth. In turn this will slow the input to Mercury. We thought that the balancing point will be interesting to observe.

We use a measurement system called TimeTrial [10] that can probe into Mercury and report various run time parameters. We decided that as input to the model equations will be the measured (empirical) arrival rate of data from disk and the fraction of elements being filtered (discarded) at each stage. We will let the model predict the inputs to all other stages, all the stage's utilizations, the mean number of elements in each queue, and the probability that a stage will assert a back-pressure signal. Of course other predictions are possible (for example the expected mean wait time in each stage) from the values we already set to calculate. The input of each stage is simply the input rate of the previous stage multiplied by the fraction of the elements that it did not filter, $\lambda_{next} = \lambda_{previous} \times P_{\bar{f},previous}$.

4.2 Notation

Referring to the general equations from queuing theory (figures 4 and 5) presented earlier we add the following notation to denote the values of a specific stage. with V representing a generic variable we use V_G to symbolize a value of a sta$G$e, V_Q to symbolize a value of a Queue, and V_S to symbolize a value of a Server.

Additional subscripts are used to denote the stages as shown in figure 7.

So, for example λ_2 denotes the mean input rate to stage 2. When more than one subscript is needed to denote a specific variable we use commas to separate the subscripts. For example $\lambda_{a,2}$ is the $a$ctual mean arrival rate to stage 2, $N_{Q,2}$ is the mean number of elements in the Queue of stage 2, $N_{S,2}$ is the mean number of elements in the Server of stage 2, and $N_{G,2}$ is the mean number of elements in sta$G$e 2. Note that G always denotes values of the queue and server summed

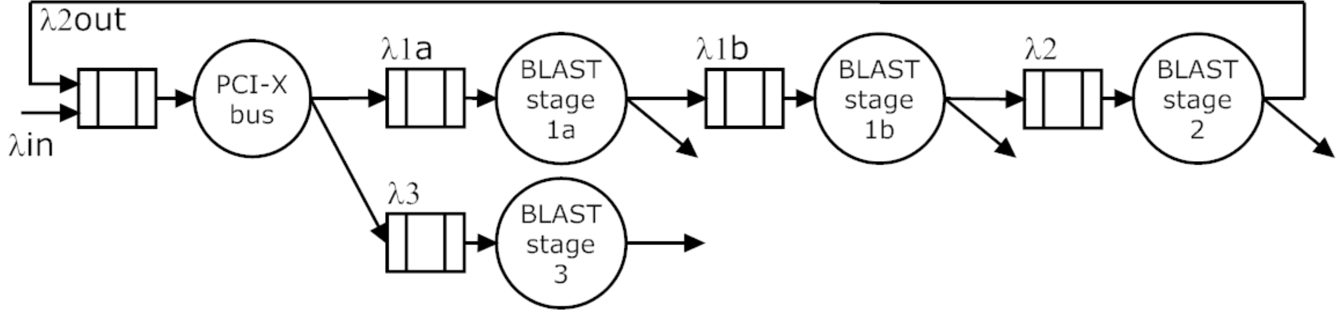

Figure 6. Queuing model of Mercury

Stage	Description
pci	PCI data bus
1a	Stage 1a, Bloom filter
1b	Stage 1b, Hash lookup
2	Stage 2, Ungapped extension
3	Stage 3, Gapped extension

Figure 7. Stages list of subscripts

Variable	Meaning
C	Clock frequency
F	Unit conversion Factor
M	Multiplier
$P_{\bar{f}}$	Probability of not filtering an element
R	Rate

Figure 8. Common variables

together.

Lastly we use the common variables depicted in figure 8 repeatedly in this work.

4.3 Calculations

4.3.1 Conversion factors

To be able to reason about each server using its native data elements we need to use conversion factors to turn each incoming data element to the native data element of the current stage. The conversion factors corresponding to the current design of Mercury BLAST appear in figure 9. A complete discussion on the use of conversion factors appears below (4.3.3).

4.3.2 TimeTrial values

The TimeTrial values used in the calculations that follow in this paper are presented in figure 10.

Stage	Factor	Value
pci	F_{pci}	1 pciElement/8 bytes
1a	$F_{1,1a}$	1 1a-word/1 pciElement
	$F_{2,1a}$	16 wmers/1 1a-word
1b	F_{1b}	1 seed/1 wmer
2	F_2	1 HSP/1 wmer

Figure 9. Mercury's native conversion factors

4.3.3 The input

For each stage the entry point is the mean arrival rate. It is a measured quantity only for the PCI stage and calculated by the model for all subsequent stages. What is directly calculated is the mean output rate of a given stage, $S-1$. By the principle of conservation of flow the output of stage $S-1$ is equal to the input of the stage S. The subscript S is replaced with the stage identifications shown in figure 7. For example the mean input rate of stage 1a is equal to the mean output rate of the PCI stage towards 1a,

$$\lambda_{in,1a} = \lambda_{pci \to 1a}.$$

The modeling of each stage starts with its input rate $\lambda_{in,S}$. To be useful for design space exploration we want to allow the input rate to be manually changed. Having this feature as part of the suggested set of equations allow the designer, once a model of a working system was obtained, to explore what will be the effects of changing the flow into the system. For this purpose we introduce two additional variables that are not used in queuing theory. The *m*anual *o*verdrive of the input rate $\lambda_{mo,S}$, and the input rate *a*fter (including) the *m*anual *o*verdrive which is the sum of the true input rate and the manual overdrive

$$\lambda_{amo,S} = \lambda_{in,S} + \lambda_{mo,S}.$$

Variable	Meaning
$1a_{in,active}$	Number of clock cycles that stage 1a is active
$1a_{in,total}$	Number of clock cycles that the system was running
$1b_{in,active}$	Number of clock cycles that stage 1b is active
$1b_{in,total}$	Number of clock cycles that the system was running
$1b_{in,median\%}$	Fraction of cycles that stage 1b is active (this is equal to $\frac{1b_{in,active}}{1b_{in,total}}$).
$SRAM_{rd,active}$	Number of clock cycles that the SRAM is accessed
$SRAM_{rd,median\%}$	Fraction of cycles that the SRAM is active
$PFP_{median\%}$	Probability that a seed returned from the Bloom filter is a false positive

Figure 10. TimeTrial variable names

The elements arriving into the current stage are data structures, or data elements that are native to the sending stage. As we present the utility of queuing theory to design space exploration we urge the designer to reason about each stage using data elements native to the stage rather than the alternate approach in queuing of normalizing all stages to some common data element that will be used throughout. This adds a little bit of algebra at the entry to each stage, but allows discourse that is understood by all designers and users of the stage. More importantly is allows much simpler reasoning as we will see below. To support this feature we use one or more conversion factors $F_{n,S}$ as shown in 9. Often only one conversion rate is needed and its index n is omitted. We calculate the $a$ctual mean arrival rate in units native to the stage as

$$\lambda_{a,S} = \lambda_{amo,S} \times (F_{1,S} \times ... \times F_{n,S}) .$$

Finally, prior stages can try to push more data than the current stage can handle. In a M/M/1/K queuing model these data elements will be dropped. In many real systems the up-stream stage will assert a back-pressure signal at the right time to stop prior stages from sending more elements. We refer to this hypothetical rate as offered input rated and denote it $\lambda_{o,S}$. The $o$ffered mean arrival rate of an M/M/1/K stage is calculated using $P_{K,S}$, the probability that the stage has run out of space for data elements (see figure 5) as follows

$$\lambda_{o,S} = \frac{\lambda_{a,S}}{1 - P_{K,S}}$$

Following is an example. Suppose that for a certain run of Mercury the output rate of the PCI stage toward stage $1a$ is 90 $^{\text{MpciElements}}/_{\text{s}}$ (where MpciElements stands for 10^6 PCI Elements). We manually overdrive the stage with additional input of 10 $^{\text{MpciElements}}/_{\text{s}}$. Stage 1a's native data structure is the 1a-word. By design the 1a-word size is equal to a PCI Element and there are 16 w-mers in each 1a-word. Lets calculate the various rates for this stage:

The mean input rate into the stage is simply the mean output rate of the previous stage

$$\begin{aligned} \lambda_{in,1a} &= \lambda_{pci \to 1a} \\ &= 90 \ ^{\text{MpciElements}}/_{\text{s}}. \end{aligned}$$

The mean input rate after (including) manual overdrive is

$$\begin{aligned} \lambda_{amo,1a} &= \lambda_{in,1a} + \lambda_{mo} \\ &= (90 \ ^{\text{MpciElements}}/_{\text{s}}) + (10 \ ^{\text{MpciElements}}/_{\text{s}}) \\ &= 100 \ ^{\text{MpciElements}}/_{\text{s}}. \end{aligned}$$

The actual mean input rate, expressed in the stage's natural data element units is

$$\begin{aligned} \lambda_{a,1a} &= \lambda_{amo,1a} \times F_{1,1a} \times F_{2,1a} \\ &= (100 \ ^{\text{MpciElements}}/_{\text{s}}) \times (1 \ ^{\text{1a-word}}/_{\text{pciElement}}) \\ &\quad \times (16 \ ^{\text{wmers}}/_{\text{1a-word}}) \\ &= 1600 \ ^{\text{Mwmer}}/_{\text{s}}. \end{aligned}$$

In a M/M/1/K queuing model the offered mean input rate can be higher than the actual mean input rate because a stage can be offered more elements than it can process. In the model these data elements are lost. In many real systems they are blocked from arriving to the stage. The offered and the actual are related to each other by the following system of equations (this system can be solved for example by substituting the second equation into the first):

$$\begin{aligned} P_{K,1a} &= \frac{(1 - \rho_{o,1a})(\rho_{o,1a})^K}{1 - (\rho_{o,1a})^K} \\ \rho_{o,1a} &= \frac{\rho_{a,1a}}{1 - P_{K,1a}} \end{aligned}$$

We can determine the actual stage utilization, $\rho_{a,1a}$, using the value for μ_{1a}. The calculation of a general μ

is shown in section 4.3.5. Using our example data this turns out to be

$$\begin{aligned}\rho_{a,1a} &= \frac{\lambda_{a,1a}}{\mu_{1a}} \\ &= \frac{1600 \ ^{\text{Mwmer}}/_{\text{s}}}{2128 \ ^{\text{Mwmer}}/_{\text{s}}} \\ &= 0.75\end{aligned}$$

Suppose that after solving the above system of equations we find that the probability of stage 1a running out of space is $P_{K,1a} = 0.20$, then the offered input rate will be

$$\begin{aligned}\lambda_{o,1a} &= \frac{\lambda_{a,1a}}{1 - P_{K,1a}} \\ &= \frac{1600 \ ^{\text{Mwmer}}/_{\text{s}}}{1 - 0.20} \\ &= 2000 \ ^{\text{Mwmer}}/_{\text{s}}.\end{aligned}$$

4.3.4 Actual and Offered values in calculations

For clarity we remind the reader that there are two server utilizations. The actual utilization, $\rho_{a,S}$, and the offered utilization, $\rho_{o,S}$. The probabilities and the expected mean number of elements equations shown in figure 5 must be calculated using the offered values.

4.3.5 Calculating the server service rate

Each server capacity is calculated from first principles, that is, from knowledge of how the system works. Not every aspect of the logic needs to be taken into account, rather the key dominating factors and the factors we wish to reason about. Each server service rate calculation starts with the number of data elements the server can process in each cycle, R_S, and the stage clock speed C_S. This view of the server's service rate is deliberately simple and does not depict all of the inner-working of its logic. However, the main thrust of our hypothesis was simplicity. We opted to give it a chance in order to evaluate the value of our assertions. As will be clear below for each stage, either key dominating factors or characteristics of the server that we want to reason about were added to the service rate calculation.

The PCI service rate was determined by empirical knowledge. The FPGA card that Mercury is implemented on is a PCI-X card. From the PCI-X specification we know that it has a theoretical maximum rate of 1064 $^{\text{MB}}/_{\text{s}}$. In our observations its effective rate was about 90% of the theoretical maximum, so we have given it a mean service rate $\mu_{pci} = 900 \ ^{\text{MByte}}/_{\text{s}}$.

For stage 1a the simple definition above was sufficient. From first principles we know that its processing rate $R_{1a} = 16 \ ^{\text{wmers}}/_{\text{clk}}$ and its clock rate is $C_{1a} = 133$ MHz. So its or service rate is

$$\begin{aligned}\mu_{1a} &= R_{1a} \times C_{1a} \\ &= (16 \ ^{\text{wmers}}/_{\text{clk}}) \times (133 \text{ MHz}) \\ &= (16 \ ^{\text{wmers}}/_{\text{clk}}) \times (133 \ ^{\text{Mclk}}/_{\text{s}}) \\ &= 2128 \ ^{\text{Mwmer}}/_{\text{s}}.\end{aligned}$$

Stage 1b checks all seeds that the Bloom filter determined are matches against hash tables. As discussed in the Mercury BLAST section the Bloom filter can return some false positives. These are not real matches between the two DNA strands. The hash tables include all the w-mers of one of the species, but checking against them is more expensive (than the Bloom filter) computationally. So only the w-mers not filtered by Bloom are verified against them, achieving in essence two objectives. Fast reduction by the Bloom filter of the number of seeds that need to be checked and proceeding only with true matches after the hash lookup (this approach yields both speed & correctness). From knowledge of the design we know that stage 1b can make 1 SRAM read per clock ($R_{1b} = 1 \ ^{\text{seeds}}/_{\text{clk}}$) and its clock speed is $C_{1b} = 133$ MHz. We also know due to the nature of hash tables that they will have collisions. When a collision occurs we could not expect that a seed will be checked against the tables in one clock cycle. TimeTrial did not report the mean (average) number of SRAM reads per seed, but we calculate it indirectly by dividing the number of clock cycles that the SRAM read signal was active by the number of clock cycles that stage 1b was active, both reported by TimeTrial. This became stage 1b multiplier

$$\begin{aligned}M_{1b} &= \text{Average \#lookups in SRAM per seed} \\ &= \frac{SRAM_{rd,active}}{1b_{in,active}}.\end{aligned}$$

For the runs we observed so far this value was around 1.06, a excitingly low number. This suggests that the hash tables organization (the hash function) is very good. With this value as an example, the mean service rate of stage 1b calculates to be

$$\begin{aligned}\mu_{1b} &= \frac{R_{1b}}{M_{1b}} \times C_{1b} \\ &= \frac{1 \ ^{\text{seeds}}/_{\text{clk}}}{1.06} \times 133 \text{ MHz} \\ &= 125.47 \ ^{\text{Mseed}}/_{\text{s}}\end{aligned}$$

For the real Mercury system the service rate of stage 1b was more complicated to obtain. A detailed account can be found in section 6.1.

Stage 2 was simpler to model. Its processing rate $R_2 = 1\ ^{\mathrm{HSP}}/_{\mathrm{clk}}$, its clock $C_2 = 133$ MHz. Its mean service rate is therefore

$$\mu_2 = R_2 \times C_2 = 133\ ^{\mathrm{MHSP}}/_{\mathrm{s}}.$$

There are infinitely many possible systems and so infinitely many servers that can be modeled. Each server should be modeled according to what the designer knows about the server behavior from first principles or from the specifications. The important characteristics to include are those that the designer would like to explore their design alternatives, or reason about, and the dominating factors. We hope that the success of this work will compel designers to take this approach.

4.3.6 The Output

The output is calculated as the product of the input rate and the fraction of the elements that are not filtered by the stage.

Since the PCI bus does not filter any elements, its output rate is simply the sum of its two inputs, the primary input to the system λ_{in} (the data stream from disk) and the output of stage 2 toward stage 3

$$\lambda_{pci \to 1a} = \lambda_{in} + \lambda_{2 \to 3}.$$

As another example the output rate of stage 1a towards stage 1b is stage 1a mean input rate multiplied by the probability that an element processed by stage 1a will not be filtered out

$$\lambda_{1a \to 1b} = \lambda_{1a} \times P_{\bar{f},1a}.$$

All the other stages follow the same equation, for a stage S and its subsequent stage T we have

$$\begin{aligned}\lambda_{S \to T} &= \lambda_S \times P_S[not\ filtering\ an\ element] \\ &= \lambda_S \times P_{\bar{f},S}.\end{aligned}$$

5 M/M/1/K model of Mercury

Once the initial model is developed it needs to be verified empirically. In particular this is true for modules with algorithmic behavior which is not clearly known or might be oversimplified during the modeling process. It was encouraging for us, on the path of validating our hypothesis, to see that process in action. In particular because the results suggested that even modeling by first principles knowledge of the stage behaviors (as opposed to exact module description that is closer to the specification of the system) came very close to the empirical data.

Parameter	Value for Run 1	Value for Run 2
λ_{in}	895 $^{\mathrm{MB}}/_{\mathrm{s}}$	722 $^{\mathrm{MB}}/_{\mathrm{s}}$
K_{1a}	130 wmers	130 wmers
K_{1b}	600 seeds	600 seeds
K_2	10 HSPs	10 HSPs
μ_{pci}	900 $^{\mathrm{MB}}/_{\mathrm{s}}$	900 $^{\mathrm{MB}}/_{\mathrm{s}}$
μ_{1a}	2130 $^{\mathrm{Mwmers}}/_{\mathrm{s}}$	2130 $^{\mathrm{Mwmers}}/_{\mathrm{s}}$
μ_{1b}	128 $^{\mathrm{Mseeds}}/_{\mathrm{s}}$	50.0 $^{\mathrm{Mseeds}}/_{\mathrm{s}}$
μ_2	133 $^{\mathrm{MHSP}}/_{\mathrm{s}}$	133 $^{\mathrm{MHSP}}/_{\mathrm{s}}$
$P_{\bar{f},pci}$	1.000	1.000
$P_{\bar{f},1a}$	0.017 8	0.034 6
$P_{\bar{f},1b}$	0.877	0.765
$P_{\bar{f},2}$	0.000 204	0.000 350

Figure 11. Inputs to M/M/1/K model

5.1 Parameters to the model

Figure 11 presents the inputs to the model. Their explanation follows:

The mean input rate to the system λ_{in} was not available directly and was calculated from TimeTrial's other reported data. Namely, the fraction of the cycles that data elements came into stage 1a multiplied by the transfer rate of the PCI bus

$$\lambda_{in} = \frac{1a_{in,active}}{1a_{in,total}} \times 1064\ ^{\mathrm{MB}}/_{\mathrm{s}}.$$

The number of elements that a stage can hold (the queue length), K_S, is a constant determined from knowledge of the design.

The server's service rates, μ_S, were calculated as explained in section 4.3.5 (the concepts) and 6.1 (taking into account some particulars of Mercury). Note for example that the PCI service rate is constant, but some stages can have data dependent service rate characteristics as can be seen from stage 1b runs 1 and 2. A complete explanation of the reason for this behavior and how the values in figure 11 were obtained appears in 6.1.

The non-filtering probabilities, $P_{\bar{f},S}$, are obtained from TimeTrial (for TimeTrial this is a measure of the fraction of the elements that were not filtered by the stage).

5.2 Results

In figure 12 we compare the model predictions to the empirical results obtained by TimeTrial and we explain the results below:

The values $\rho_{a,pci}$ and $N_{Q,1a}$ were not available from TimeTrial. Refer to the discussion in 6.3 for detailed

Parameter	Model prediction	Empirical measurement	Model prediction	Empirical measurement
Run 1			Run 2	
$\lambda_{a,1a}$	1790 Mwmers/s	1790 Mwmers/s	1440 Mwmers/s	1440 Mwmers/s
$\lambda_{a,1b}$	31.9 Mseeds/s	31.9 Mseeds/s	50.0 Mseeds/s	50.0 Mseeds/s
$\lambda_{a,2}$	28.0 MHSPs/s	28.0 MHSPs/s	38.3 MHSPs/s	38.3 MHSPs/s
$\rho_{a,pci}$	0.995	N/A	0.802	N/A
$\rho_{a,1a}$	0.841	0.848	0.679	0.679
$\rho_{a,1b}$	0.250	0.233	1.00	0.927
$\rho_{a,2}$	0.210	0.207	0.288	0.288
$N_{Q,1a}$	4.47 wmers	N/A	1.43 wmers	N/A
$N_{Q,1b}$	0.08 seeds	approaching 0 seeds	458 seeds	580 seeds
$N_{Q,2}$	0.06 HSPs	1.2 HSPs	0.12 HSPs	1.7 HSPs

Figure 12. Results of M/M/1/K model

explanation about how the utilization's empirical values were obtained.

We see that most data points are surprisingly very well predicted. The single value that is not exactly on the mark is the number of elements in the queue of 1b (and that discrepancy exists only in the second run). Possibly some of the discrepancy can be attributed to the difference in scope of $N_{Q,1b}$ in the model and TimeTrial. In the model this represents the number of elements in all the queues and sub-servers internal to stage 1b. But to obtain the empirical value we added the number of elements in two such queues, the elements in the queue at the input to 1b and the elements waiting for SRAM lookup

$$N_{Q,1b} = (\#elements\ in\ wmer\ FIFO) + (\#elements\ in\ SRAM\ FIFO)\,.$$

6 M/M/1/K model of Mercury with decoupled stage 1b

In an effort to model stage 1b a little bit more deeply we decided to decouple it into two modules as shown in figure 13. This effort is articulated next.

6.1 Decoupling stage 1b

It is known that stage 1b does not have a straight forward behavior. Each arriving seed can be a false positive generated due to the probabilistic behavior of stage 1a's Bloom filter, in which case it will not be found in the hash tables by stage 1b. If it is not a false positive (so it's a valid seed) the lookup in the primary hash table will return a result that can have the following options:

(1) A seed can match 2 or less positions in the query DNA string.

(2) A seed can match more than 2 positions in the query DNA string.

(3) In either case the primary hash table can indicate that there is a hash collision in that position.

Stage 1b memory was designed such that seeds with 2 or less matches are stored in a hash table called primary and the rest are in a table called duplicates. We decouple stage 1b into two stages. Stage 1b-primary (denoted $1bp$) with the logic to look for a seed in the primary hash table and stage 1b-duplicate (denoted $1bd$) responsible for looking for a seed in the duplicate hash table.

For case (1) the hash lookup process is fully pipelined. One of the simplification principles we used in modeling is that pipelined behavior can be abstracted away by modeling the service time as the interval between successive inputs that a server can accept. This was used in the M/M/1/K model for all stages and so far proved to be a valid abstraction, at least for the single experiment we presented earlier. Using this principle we can say that the SRAM returns an answer in each clock cycle. We continue to model stage 1b-primary in this way.

Cases (2) and (3) go into a shared queue (which is also shared with case (1) seeds, a matter that further complicates stage 1b behavior). That queue feeds an unpipelined server that returns an answer within about 60 clock cycles. If there are more than 5 matches, 120 clock cycles, and so on.

Continuing our desire to abstract complicated behavior away (within reason) while maintaining the goal of constructing a reasonably correct model, we decided to not count exactly how many seeds fall into cases (2) and (3). We did not have access to this value using the current features of TimeTrial even if we wanted to include it in the model. Instead we realized that stage 1b service rate is heavily dependent on the data

because two DNA strings which share more common sub-strings will have more matches and so will fall into case (2) more often. Case (3) is dependent on the properties of the hashing and will be abstracted away unless the model will be proved to need this refinement. Since the SRAM where the hash tables are stored is a shared resource for all SRAM lookups, a lookup in the primary table will wait for such a cycle to complete. So in essence case (1) seeds could also wait 60 clock cycles instead of the expected 1 lookup per clock.

In the decoupled model of stage 1b we are clear about when the round trip latency (the 60 clocks) is limiting the rate in which stage 1b as a whole can proceed. In the coupled model we presented earlier we did not account for this behavior. When there are relatively few duplicates (that is very few 60 clocks cycles) the measured utilization from TimeTrial should be consistent with the service rate of the **input** to this stage (the service rate of 1b primary). In other words when an SRAM lookup takes about one clock cycle, it does not delay 1b input. That happens when 1b is not saturated due to many duplicate checks (very few case (2) seeds, ignoring case (3) which should have fairly low probability if the hashing function is well designed). Coincidentally Run 1 is such a run, the M/M/1/K model suggested a utilization of $\rho = 0.25$ and close to 0 elements in the queue ($N_{Q,1b}$ is approaching 0). The empirical results we presented earlier validated these predictions for this run. We will therefore use this run as the lower-end bound of stage 1b's capacity and expect that stage 1bp will not be limited by stage 1bd for this run. In this case the mean service rate of stage 1b is expected to stay as it was calculated before

$$\begin{aligned}\mu_{1b} =& \frac{R_{1b}}{M_{1b}} \times C_{1b} \\ =& \frac{1 \ ^{\text{seeds}}/_{\text{clk}}}{M_{1b}} \times 133 \ \ \text{MHz} \\ =& \frac{133}{M_{1b}} \ ^{\text{Mseed}}/_{\text{s}}.\end{aligned}$$

On the other end of the scale for the stress on stage 1b, Run 2 seems to saturate it. The model predicts utilization $\rho = 1.00$ and almost a full queue with $N_{Q,1b} = 460$. The empirical numbers validate these values. But TimeTrial reports that the SRAM utilization for this run is only about 0.40. This suggest that the maximum capacity of the stage is capped at 40% for this run. The clock cycles being spent on the duplicate lookups delay the whole stage. If so, then the maximum service rate of this module is not as it was calculated before, rather we expect that its top performance will be capped at about 0.4 of its theoretical maximum rate for runs with many duplicates (many seeds with more than 2 matches between the two DNA strands). To account for this situation we add a second factor which we obtain from the empirical mean utilization of the SRAM, now treated as the effective maximum

Parameter	Value for Run 1	Value for Run 2
λ_{in}	895 $^{\text{MB}}/_{\text{s}}$	722 $^{\text{MB}}/_{\text{s}}$
K_{1a}	130 wmers	130 wmers
K_{1bp}	130 seeds	130 seeds
K_{1bd}	500 seeds	500 seeds
K_2	10 HSPs	10 HSPs
μ_{pci}	900 $^{\text{MB}}/_{\text{s}}$	900 $^{\text{MB}}/_{\text{s}}$
μ_{1a}	2130 $^{\text{Mwmers}}/_{\text{s}}$	2130 $^{\text{Mwmers}}/_{\text{s}}$
μ_{1bp}	128 $^{\text{Mseeds}}/_{\text{s}}$	50.0 $^{\text{Mseeds}}/_{\text{s}}$
μ_{1bd}	128 $^{\text{Mseeds}}/_{\text{s}}$	50.0 $^{\text{Mseeds}}/_{\text{s}}$
μ_2	133 $^{\text{MHSP}}/_{\text{s}}$	133 $^{\text{MHSP}}/_{\text{s}}$
$P_{\bar{f},pci}$	1	1
$P_{\bar{f},1a}$	0.017 8	0.034 6
$P_{\bar{f},1bp}$	0.923	0.892
$P_{\bar{f},1bd}$	0.950	0.857
$P_{\bar{f},2}$	0.000 204	0.000 350

Figure 14. Inputs to M/M/1/K model with the decoupled stage 1b

$$M_{2,1b} = SRAM_{rd,median\%} = 0.406.$$

The mean number of lookups in SRAM per seed, M_{1b}, is still a run time parameter, we just add to it a subscript (we index it) to differentiate it from this second multiplier we just added to the model. We set $M_{1,1b} = M_{1b}$ and calculate the maximum effective service rate of stage 1b to be:

$$\begin{aligned}\mu_{1b} =& \frac{R_{1b}}{M_{1,1b}} \times C_{1b} \times M_{2,1b} \\ =& \frac{1 \ ^{\text{seeds}}/_{\text{clk}}}{M_{1,1b}} \times 133 \ \ \text{MHz} \times 0.406 \\ =& \frac{50.0}{M_{1,1b}} \ ^{\text{Mseed}}/_{\text{s}}.\end{aligned}$$

6.2 Parameters to the model

Figure 14 presents the inputs to the decoupled stage 1b model.

With these stages there are a few changes in the parameters to the model. The size of the queue of stage 1b primary, $1bp$, is 130 data elements and it represents the queue size at the entry to this stage. The size of the queue of stage 1b duplicate is the queue of all the elements waiting to for a hash lookup. We note that

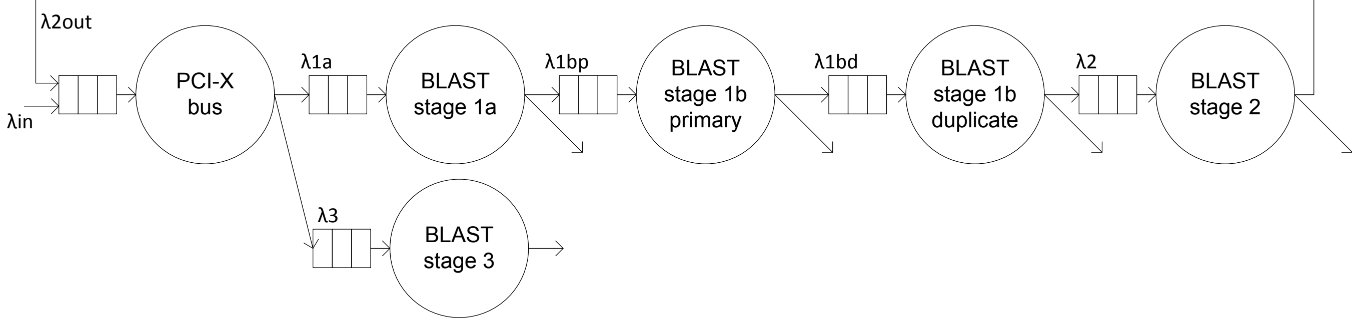

Figure 13. Queuing model of Mercury with decoupled stage 1b

this is a simplified view of this queue because it is, in fact, shared between stage $1bp$ and $1bd$. Its size is 500 seeds.

The servers service rate, μ_{1bp} and μ_{1bd}, were calculated as explained in 6.1 (these results were also used in section 5).

There is a significant change in how the non-filtering fractions for stage 1b are determined. The non-filtering fraction $P_{\bar{f},1bp}$ is the probability that a seed will continue to stage 1_{bd}, or in other words it is the probability that a seed returned from the Bloom filter is not a false positive (is a real seed). The probability that it is a false positive is $1 - P_{\bar{f},1bp}$. It can be calculated from the results as follows

$$P_{\bar{f},1bp} = 1 - PFP_{median\%}.$$

The non-filtering fraction $P_{\bar{f},1bd}$ is the probability that a seed will continue to stage 2, that is, it is the probability that a seed will be found in the duplicate hash tables. Since the probability that a seed passes the whole stage 1b (and so continues to stage 2) is the product of the probability that it is not a false positive and the probability that is is found in the hash tables

$$P_{\bar{f},1b} = P_{\bar{f},1bp} \times P_{\bar{f},1bd}$$

changing sides in this equation we have the probability that a seed will be found in the hash tables

$$P_{\bar{f},1bd} = \frac{P_{\bar{f},1b}}{P_{\bar{f},1bp}}.$$

6.3 Results

With these changes to the model we obtained the results presented in figure 15.

The values marked as N/A were Not Available from TimeTrial. We disclose that the empirical values were not directly reported for the saturated run by TimeTrial. We also needed to use a different calculation for this value for the two runs. For the not-saturated run, Run 1, the empirical value was taken directly from the reported TimeTrial value

$$\rho_{a,1b,empirical} = 1b_{in,median\%}.$$

But for the saturated run, Run 2, it was calculated from the utilization of stage 1b divided by the utilization of the SRAM

$$\rho_{a,1b,empirical} = \frac{1b_{in,median\%}}{SRAM_{rd,median\%}}.$$

We expect the empirical values to correspond to these calculations because (as discussed earlier) when stage 1b is not saturated its overall performance is not impeded by the long (60 clock cycles) round trip of each duplicate lookup. In this case the effective service rate is close to the clock speed. But when there are data elements that require a lookup in the duplicate hash tables, because stage 1bd is not pipelined in essence the data elements that requires a duplicate lookup slows the 1b as a whole by a factor of 60 clock cycles. This effective service rate can be approximated by the count of how many times the stage was active over how many times the SRAM was active as calculated in equation 4.3.5. This is only an approximation (but seems to be a very good one) because the SRAM is active for the primary reads as well. We do not suggest at this point the type of correlation (linear or not) between the two extremes, the non-saturated and saturated run.

As can be seen from table 15 the model predicts very well. This is a second promising result. The only discrepancy is the number of elements in the queue of stage 1bd for run 2 which can be explained by the fact that the data dependent deficiency factor, $M_{2,1b}$, is approximately equal to $SRAM_{rd,median\%}$, but it is not exactly that. In fact for run 2 it was calculated and used in the reported results to be equal to 2.464, but if we change it to 2.761 (we change only the factor for stage 1b duplicate, the factor for stage 1b primary

Parameter	Model prediction	Empirical measurement	Model prediction	Empirical measurement
	Run 1		Run 2	
$\lambda_{a,1a}$	1790 Mwmers/s	1790 Mwmers/s	1440 Mwmers/s	1440Mwmers/s
$\lambda_{a,1bp}$	31.9 Mseeds/s	31.9 Mseeds/s	50.0 Mseeds/s	50.0Mseeds/s
$\lambda_{a,1bd}$	29.5 Mseeds/s	N/A	44.6 Mseeds/s	N/A
$\lambda_{a,2}$	28.0 MHSPs/s	28.0 MHSPs/s	38.3 MHSPs/s	38.3MHSPs/s
$\rho_{a,pci}$	0.995	N/A	0.802	N/A
$\rho_{a,1a}$	0.841	0.848	0.679	0.679
$\rho_{a,1bp}$	0.250	0.233	1.000	0.927
$\rho_{a,1bd}$	0.230	N/A	0.892	N/A
$\rho_{a,2}$	0.210	0.207	0.288	0.288
$N_{Q,1a}$	4.47 wmers	N/A	1.43 wmers	N/A
$N_{Q,1bp}$	0.08 seeds	approaching 0 seeds	107 seeds	115 seeds
$N_{Q,1bd}$	0.07 seeds	approaching 0 seeds	7.40 seeds	460 seeds
$N_{Q,2}$	0.06 HSPs	1.2 HSPs	0.12 HSPs	1.7 HSPs

Figure 15. Results of M/M/1/K model for the decoupled stage 1b

was kept at 2.464), then $N_{Q,1bd}$ is over 400, close to the empirical value. The value of 2.762 is already beyond the model range, suggesting that stage 1bd is operating very close to its saturation point.

7 M/M/1 model of Mercury

We have shown that M/M/1/K models work and justify our hypothesis, but can we do better? Can we use an even simpler queuing model to correctly model a real streaming application?

An M/M/1 queuing model has infinite queues so no data elements are ever lost. However, real systems do have a limited number of elements that they can hold in their stages. Probability theory in this case allows us to easily calculate the cumulative distribution. We are planning to take advantage of this situation by adding an additional modeled prediction, the probability of asserting back-pressure. We claim that this probability is equal to the probability of the stage having more than the number of elements that it can physically hold in its queue and server.

To answer the question of the utility of an M/M/1 model we go back to the original queuing model of Mercury as shown in figure 6.

7.1 Parameters to the model

The μ and ρ calculations are the same as for M/M/1/K models. The K are not used in queuing theory of M/M/1 models, but they can effectively be used for back-pressure probability calculation as explained above. Figure 16 show the inputs to the model.

Parameter	Value for Run 1	Value for Run 2
λ_{in}	895 MB/s	722 MB/s
K_{1a}	130 wmers	130 wmers
K_{1b}	600 seeds	600 seeds
K_2	10 HSPs	10 HSPs
μ_{pci}	900 MB/s	900 MB/s
μ_{1a}	2130 Mwmers/s	2130 Mwmers/s
μ_{1b}	128 Mseeds/s	50.0 Mseeds/s
μ_2	133 MHSP/s	133 MHSP/s
P_{pci}	1	1
P_{1a}	0.017 8	0.034 6
P_{1b}	0.877	0.765
P_2	0.000 204	0.000 350

Figure 16. Inputs to M/M/1 model

7.2 Results

Figure 17 displays the results.

We observe that there is a close match between almost all of the model predictions and the empirical measurements. The input rates exactly match up. The servers utilization fit very nicely. These two parts are not surprising. The input rates depend the output of a prior stage and the filtering fraction, and the server utilization are based primarily on mean flow rates, so both are insensitive to the probability distributions.

Turning to the queue occupancies, the only important discrepancy is the queue associated with stage 1b in run 2. Here, the high server utilization indicates that this server is the performance limiting bottleneck in the application. The physical queue is of length 600 entries so the empirical queue occupancy cannot grow larger

Parameter	Model prediction	Empirical measurement	Model prediction	Empirical measurement
	Run 1		Run 2	
λ_{1a}	1790 Mwmers/s	1790 Mwmers/s	1440Mwmers/s	1440Mwmers/s
λ_{1b}	31.9 Mseeds/s	31.9 Mseeds/s	50.0Mseeds/s	50.0Mseeds/s
λ_2	28.0 MHSPs/s	28.0 MHSPs/s	38.3MHSPs/s	38.3MHSPs/s
ρ_{pci}	0.995	N/A	0.802	N/A
ρ_{1a}	0.841	0.848	0.679	0.679
ρ_{1b}	0.250	0.233	1.000	0.927
ρ_2	0.210	0.207	0.288	0.288
$N_{Q,1a}$	4.47 wmers	N/A	1.43 wmers	N/A
$N_{Q,1b}$	0.08 seeds	approaching 0 seeds	7470 seeds	580 seeds
$N_{Q,2}$	0.06 HSPs	1.2 HSPs	0.12 HSPs	1.7 HSPs
$P_{BP,1a}$	0.0	0.0	0.0	0.296
$P_{BP,1b}$	0.0	0.0	0.923	0.604
$P_{BP,2}$	0.0	0.0	0.000	0.0

Figure 17. Results of M/M/1 model

than that. Both the model and the empirical results are indicating that the queue will fill; however, the infinite queue capacity in the model is not capped by the length of the physical queue since this is a $M/M/1/\infty$ model. The physical system is working, so we expect that TimeTrial will return a number slightly smaller than the queue length so TimeTrial will not be able to find more than the maximum number of elements that can fit in this stage. The model will show a queue that is full because it does not take into account the physical size of the queue. As the tables shows us, this is indeed the case. To further support this explanation we can refer to the probability that the queue occupancy is greater than the actual capacity of the stage. For run 2 the table shows that to be 0.923, a high probability as we expected.

The difference between the model's probability of asserting back pressure by stage 1, $P_{BP,1a}$, and the empirical value is due to the fact that this run is not really saturating stage 1a, but stage 1a is asserting a back pressure signal due to the fact that when stage 1b assert its back pressure towards 1a, and so 1a must delay its own input. This cannot be predicted by the model because feedback are not taken into account.

Having predicted both the server utilizations and the queue occupancies implies that the assumptions present in the M/M/1 queuing models do not inordinately impact the quality of the model for these two runs. These two runs were selected because they represent two distinct execution circumstances. Run 1 lightly taxes the system while run 2 heavily taxes at least stage 1b.

8 Conclusions

We have illustrated the use of simple queuing models to correctly describe the behavior of high performance streaming applications. Our chief hypothesis that one can use seemingly inappropriate queuing models, in essence the wrong probability distribution, to correctly model real systems seems to have at least some merit. Additional verification is needed both with different data sets for Mercury BLAST as well as using the modeling approach suggested here with other systems. The M/M/1 and M/M/1/K models were surprisingly good at this task both for the four stage queuing model as well as for the more detailed five stage model we examined in this work.

We note that M/M/1/K models yield excellent predictions. The single discrepancy (stage 1bd for run 2) thus far is probably due to incomplete understanding on our part of the inner working of that particular stage. Lastly we note that M/M/1/K models cannot be easily used to model the probability of back pressure.

The M/M/1 model does surprisingly good as well in predicting the behavior of this real streaming application. Its equations are simpler than the M/M/1/K model and this is a good reason to use it to model applications. We point out the use of the length of the queue to correspond to the probability that a system will assert a back pressure signal. This is an additional advantage of M/M/1 models.

We have shown that when there are discrepancies, the model can assist in understanding those discrepancies. When one uses a M/M/1/K model a wrong prediction of the queue length is a good indicator that the model is beyond its predictive domain. For M/M/1

models we recommend using the probability of a back pressure as that indication.

9 Future work

Our intent is to use models of this type to guide tuning of the implementation, to propose design alternatives that will increase a system performance, or to examine the potential performance benefits achievable by exploiting alternative accelerators (e.g., graphics engines) for one or more of the pipeline stages. We also intend to perform more experiments and further substantiate the utility of using the M/M/1 and M/M/1/K models probability distributions to model real systems. They are easy to solve while providing excellent insight into the system behavior. Lastly we will look into the reasons that allow these incorrect probability distributions do such a good job in correctly predicting the behavior of streaming applications.

References

[1] A. O. Allen. *Probability, Statistics, and Queuing Theory with Computer Science applications*. Academic Press, 1978.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–10, 1990.

[3] S. F. Altschul, T. L. Madden, A. A. Schäffer, J. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: A new generation of protein database search programs. *Nucl. Acids Res.*, 25(17):3389–3402, Sept. 1997.

[4] F. Baskett, K. M. Chandy, R. R. Muntz, and F. G. Palacios. Open, closed, and mixed networks of queues with different classes of customers. *J. ACM*, 22(2):248–260, 1975.

[5] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, May 1970.

[6] J. D. Buhler, J. M. Lancaster, A. C. Jacob, and R. D. Chamberlain. Mercury BLASTN: Faster DNA sequence comparison using a streaming hardware architecture. In *Proc. of Reconfigurable Systems Summer Institute*, July 2007.

[7] P. Krishnamurthy. *Performance Evaluation for Hybrid Architectures*. PhD thesis, Dept. of Computer Science and Engineering, Washington University in St. Louis, Dec. 2006.

[8] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, A. Jacob, and J. Lancaster. Biosequence similarity search on the *Mercury* system. *Journal of VLSI Signal Processing*, 49(1):101–121, Oct. 2007.

[9] J. Lancaster, J. Buhler, and R. D. Chamberlain. Acceleration of ungapped extension in Mercury BLAST. *Journal of Microprocessors and Microsystems*, 33(4):281–289, June 2009.

[10] J. Lancaster, J. Buhler, and R. D. Chamberlain. Efficient runtime performance monitoring of FPGA-based applications. In *Proc. of 22nd IEEE Int'l System-on-Chip Conf.*, pages 23–28, Sept. 2009.

[11] H. G. Perros and T. Altiok. Approximate analysis of open networks of queues with blocking: Tandem configurations. *IEEE Trans. Softw. Eng.*, 12(3):450–461, 1986.

[12] Wikipedia. Genome — wikipedia, the free encyclopedia, 2010. [Online; accessed 8-April-2010].