

Optimization for Multi-thread Data-Flow Software

Helmut Hlavacs and Michael Nussbaumer

University of Vienna, Research Group Entertainment Computing,
Lenaugasse 2/8, 1080 Vienna, Austria
`{helmut.hlavacs,m.nussbaumer}@univie.ac.at`

Abstract. This work presents an optimization tool that finds the optimal number of threads for multi-thread data-flow software. Threads are assumed to encapsulate parallel executable key functionalities, are connected through finite capacity queues, and require certain hardware resources. We show how a combination of measurement and calculation, based on queueing theory, leads to an algorithm that recursively determines the best combination of threads, i.e. the best configuration of the multi-thread data-flow software on a given host. The algorithm proceeds on the directed graph of a queueing network that models this software. Experiments on different machines verify our optimization approach.

Keywords: Software Optimization, Performance Optimization, Multi-thread Software.

1 Introduction

The trend to many cores inside CPUs enables software engineering towards concurrency. Software is split up in atomic actions that can run in parallel. This may considerably speed up computation, but also causes extra overhead through thread coordination for both, the operation system and also the software engineer. Developing software made of several threads is much more complicated than creating an old-fashioned single thread software. One fundamental problem of the developer might be to find the right strategy for determining the optimal number of threads.

In this work we use the term host for a server hardware platform with a fixed number of cores. Nodes are specified as tasks of the queueing network software and build the data-flow graph. A thread executes node tasks. Each node can have multiple threads that are executing the nodes tasks in parallel. The overall situation is that we want to find the optimal number of threads per node, with the side constraint, that there can only be as many threads as cores available on the host, because each thread runs on a dedicated core.

This paper discusses an approach to use a combination of analytical modeling techniques and measurements to find an optimal configuration of threads on a given host by iteratively increasing the number of threads. The application area

is software following the data-flow paradigm, in our case a commercial tool that computes and persists call data from a telecom network.

Section 2 of this paper presents related work. In Section 3 an example multi-thread data-flow software is introduced, an analytical model is introduced, and theoretical optimization is explained. Section 4 presents our optimization approaches to find the optimal configuration for multi-thread data-flow software. Finally, Section 5 presents experiments conducted on two SUN machines with a real queueing network software, and compares them to a solely analytical approach.

2 Related Work

In the past there have been several approaches for adapting software to various multi-computers, in order to optimize the performance.

FFTW [1] is a free software library that computes the *Discrete Fourier Transform* (DFT) and its various special cases. It uses a planner to adapt its algorithms to the hardware in order to maximize performance. The FFTW planner works by measuring the actual run time of many different plans and by selecting the fastest one.

The project *SPIRAL* [2] aims at automatically generating high performance code for linear *Digital Signal Processing* (DSP) transforms, tuned to a given platform. SPIRAL implements a feedback-driven optimizer that intelligently generates and explores algorithmic and implementation choices to find the best match to the computer's micro architecture.

The *Automatically Tuned Linear Algebra Software* (ATLAS) [3] aims at automatically generating code that provides the best performance for matrix multiplication on a given platform by finding the cache-optimal variant.

In [4] an algorithm is developed for finding the nearly best configuration for a Web system. Up to 500 emulated clients generate traffic for various trading operations like *buy* or *sell*. Optimal configurations are found iteratively, depending on the number of threads and the cache size.

The technology *Grand Central Dispatch*¹ (GCD) by Apple enables to use multicore processors more easily. Firstly released in the Mac OS X 10.6, GCD is implemented by the library *libdispatch*². GCD is a scheduler for tasks organized in a queuing system and acts like a thread manager that queues and schedules tasks for parallel execution on processor cores.

In [5] a queueing network model with finite/single capacity queues and blocking after service discipline is used to model software architectures; more exactly the synchronization constraints and synchronous communication between software components. The information flow (trace) between components is analyzed to identify the kind of communication (fork, join) and reveal the interaction pairs among components that enables to model a queueing network. This way a performance model for a specific software architecture can be derived.

¹ <http://www.apple.com/macosx/technology/#grandcentral>

² <http://libdispatch.macosforge.org>

3 Analytical Model

The system under investigation is a commercial product called Data Flow Engine (DFE) for processing and persisting call data stemming from standard telecom networks. In an industrial cooperation, our task was to find automatic ways for analyzing the performance of the software, and adapt the software to different hardware platforms and workload situations. The software in some sense should utilize the hardware in some optimized way, for example for maximizing the throughput, but also utilize only as much hardware as needed, for example for optimizing the energy consumption if this is supported by the platform.

Data describing call state changes like calling, canceling etc. is represented by tickets or packets being sent from the telecom infrastructure to the software that subsequently receives, extracts, converts, and stores the incoming packets. The software itself follows a data-flow approach, organizing its tasks into a network of processing nodes (Fig. 1). Queues buffer the output for succeeding nodes and incoming tickets pass the graph and visit each node once. Since these nodes technically are lightweight processes (threads), they can be replicated for splitting the load. Nodes model atomic actions, but several instances of a node can exist as threads.

Currently defined nodes are:

- The *Decoder* node takes packets from its queue, extracts the data, and forwards it to the next node’s queue. Hence, the extraction can be done in parallel by several nodes. A *Decoder* can be replicated, each *Decoder* forwards the extracted data to the same queue.
- The *Converter* node takes extracted data from its queue, converts it into a format appropriate for storing and forwards it to a *Serializer* and a *Feeder* node. Thus, the extracted and formatted data is persisted always twice. Also conversion can be done in parallel by several threads of the *Converter*.
- The *Serializer* node takes data from its queue and stores it to disk. *Serializers* may write in parallel, even to different disks.
- The *Feeder* node takes data from its queue and sends it to a database. Since the database is assumed to be capable to provide a connection pool, *Feeders* may send in parallel.

All nodes require certain CPU time, memory, disk space and disk bandwidth, and network bandwidth. Thus, the benefit of replicating nodes is limited by the available hardware. Starting with an initial configuration with one thread per node, the goal is to find the optimal configuration of threads for a certain host.

The main idea is to sequentially add new threads for over-utilized nodes until an optimization goal is reached. For that reason, a queueing network, as base for an analytical model [6–9] for describing the multi-thread software, is shown in Fig. 1 as an open queueing network consisting of 4 queues. Open in the sense that jobs come from an external source, are serviced by an arbitrary number of servers inside the network and eventually leave the network. Further, the

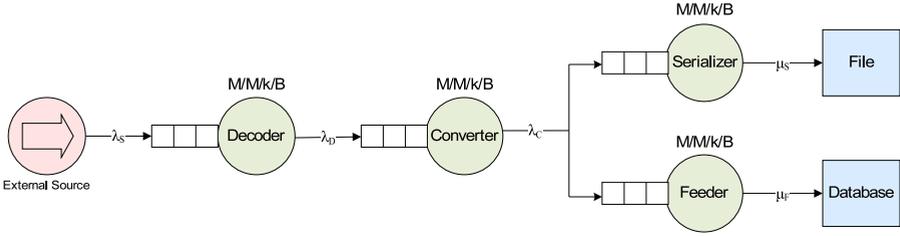


Fig. 1. Multi-thread software modeled as queuing network

suggested queuing network is aperiodic, because no job visits a server twice, the flow goes to one direction. The queuing discipline is always *First-Come, First-Served* (FCFS).

An external source sends events with rate λ_S to the M/M/k/B *Decoder* queue whereas the arrival process (M/././.) is Markovian and follows a Poisson process [10] with independent identically distributed (iid) and exponentially interarrival times $1/\lambda$, which postulates that the next arrival at $t + 1$ is completely independent from the arrival at t . The use of Poisson arrival is motivated by the fact that the workload source consists of possibly millions of independent sources (callers) who create call data independently from each other.

The service time $1/\mu$ of each server k is independent from the arrival process and iid and exponentially distributed (memoryless) with parameter μ and therefore the departure process (./M/./.) is also Markovian. The queue capacity (buffers) is defined by parameter B . An arrival reaching a full queue is blocked. This can be avoided by increasing the queue size B , decreasing the arrival rate λ , or increasing the number of servers k and so increasing the joint service rate μ . This holds also for the *Converter*, *Feeder*, and *Serializer* queues.

After the data is extracted by the *Decoder*, it is forwarded with rate λ_D to the *Converter* queue. The *Converter* then converts the data to the file and database formats and forwards it to the corresponding queues with rate λ_C . The *Feeder* sends the data to the database with rate μ_F whereas the *Serializer* writes data to disk with rate μ_S .

For each node in the graph, there is a race between λ and μ in the sense that under the assumption $\lambda \leq \mu$ for the utilization ρ and given number of servers (i.e., in our case threads) k in that node, the following must hold:

$$\rho = \frac{\lambda}{k\mu} \leq 1 \quad , \tag{1}$$

that leads to a stable node: all jobs in the node’s queue can eventually be worked out. Due to different nodes, the bottleneck is the node where $\lambda > \mu$. For finding the best configuration of the queuing network (see Fig. 1), the following performance measures [6–9] are considered for the nodes *Decoder*, *Converter*, *Serializer* and *Feeder*.

The utilization ρ from Equ. (1) of a particular node is the base for most other measures. The probability P_n of n jobs in the node is given by

$$P_n = \frac{(k\rho)^n}{n!} P_0 \quad \text{for } 0 \leq n < k \tag{2}$$

$$P_n = \frac{k^k \rho^n k!}{P} \quad \text{for } k \leq n \leq B \text{ and } B \geq k \tag{3}$$

where k is the number of servers, B the number of buffers (slots in the queue that can be set to a sufficient large number), and P_0 the probability of no jobs in the node, which for $k = 1$ is

$$P_0 = \frac{1 - \rho}{1 - \rho^{B+1}} \quad \text{for } \rho \neq 1 \tag{4}$$

$$P_0 = \frac{1}{B + 1} \quad \text{for } \rho = 1 \tag{5}$$

and for $k > 1$

$$P_0 = \left(1 + \frac{(1 - \rho)^{B-k+1} (k\rho)^k}{k!(1 - \rho)} + \sum_{n=1}^{k-1} \frac{(k\rho)^n}{n!} \right)^{-1} . \tag{6}$$

The expected number of jobs E_s in a node, for $k = 1$ is

$$E_s = \frac{\rho}{1 - \rho} - \frac{(B + 1)\rho^{B+1}}{1 - \rho^{B+1}} \tag{7}$$

and for $k > 1$

$$E_s = \sum_{n=1}^B n p_n \tag{8}$$

where the expected number of jobs in a queue E_q for $k = 1$ is

$$E_q = \frac{\rho}{1 - \rho} - \rho \frac{1 + B\rho^B}{1 - \rho^{B+1}} \tag{9}$$

and for $k > 1$

$$E_q = \sum_{n=k+1}^B (n - k) p_n \tag{10}$$

Since we have queues with finite buffers B , some traffic is blocked. The initial traffic that reaches the queueing network is not equal to the traffic that passes through the queueing network. Reduced to a single node this means that λ depends on a certain blocking probability P_b . This leads to the effective arrival rate λ'

$$\lambda' = \lambda(1 - P_b)$$

where the blocking probability $P_b = P_B$, i.e. the probability of B jobs in a node. The loss rate ϵ is

$$\epsilon = \lambda P_B \quad (11)$$

and the effective utilization ρ' is

$$\rho' = \frac{\lambda'}{k\mu}$$

again with k servers. Next, the mean response time R of a node is

$$R = \frac{E_s}{\lambda'} \quad (12)$$

and the mean waiting time W of a job in the queue is

$$W = \frac{E_q}{\lambda'} \quad (13)$$

Finally, the probability that the buffers of a node are all occupied is denoted by

$$P_k = \frac{\frac{(k\rho)^k}{k!}}{\sum_{j=0}^k \frac{(k\rho)^j}{j!}} \quad (14)$$

Configurations of the queuing network can be evaluated according to these measures. An optimization algorithm, presented in Section 4.1, determines the best configuration.

4 Optimization Approaches

The focus of our work is to find the optimal configuration of a multi-thread data-flow software for a specific host. A configuration is specified as a vector of n tuples with

$$(k_1 - \dots - k_i - \dots - k_n) \quad (15)$$

where k_i denotes the number of threads of node i . With the constraint of

$$\sum_{i=1}^n k_i \leq c \quad (16)$$

where c is the number of available (virtual) cores on a specific host. E.g.: The initial configuration of *Decoder*, *Converter*, *Serializer* and *Feeder* nodes (1-1-1-1) contains one thread per node. Optimization is done recursively by adding threads to over-utilized nodes with two optimization goals:

- Consolidation. With optimization towards consolidation a desired arrival rate is given and the tool determines the minimum number of threads required to ensure that all nodes are below a predefined utilization threshold. With a constant external arrival rate only the number of threads of an over-utilized node with utilization $\geq \text{lim}U$ ($0 < \text{lim}U < 1$) will be incremented for splitting load among available cores as evenly as possible.
- Throughput. With optimization towards throughput we start with the lowest arrival rate of one job per second and the tool increases the arrival rate step-wise until one node exceeds the predefined utilization threshold. Then the number of the corresponding threads is increased as long as the utilization is below the threshold. Again, the arrival rate will be increased and optimization goes on as long as no hardware limit is reached. With $0 < \text{ext}ARInc < 1$ the external arrival rate is increased for each new configuration by $\text{ext}ARInc$ as long as the maximum utilized node does not reach $\text{lim}U$. For the maximum possible number of threads the highest possible throughput is determined.

For calculating the best configuration of nodes, based on a host's resources (for example given by Table 1) the number of CPU cores is the upper bound for the number of nodes that can run in parallel. It is assumed that each node is executed as single thread on a dedicated core and since core sharing is ignored for now, as many nodes as free cores available are possible. Memory, disk space and speed, and network bandwidth are shared by all nodes and are the constraints for optimization. When a configuration exceeds given resources, the algorithm terminates.

Table 1. Resources of a hypothetical host

Resource Type	Quantity
CPU cores (#)	32
Memory (MB)	32000
Disk space (MB)	100000
Disk speed (MB/s)	30
Network (Mbit/s)	100

4.1 Optimization Algorithm

Our first approach in finding the optimal configuration of threads for the proposed queuing network software was to implement the aforementioned performance metrics. We therefore created a Java³ test tool that calculates all the necessary metrics and recursively adds threads to over-utilized nodes.

The optimization process of the calculation module can be described as follows:

³ <http://java.sun.com>

1. The calculation module uses parameters like the external arrival rate, the service rates for each node, the queue sizes for each node and the host's hardware details.
2. The calculation module starts with an initial configuration of one thread per node.
3. The calculation module calculates all the performance metrics mentioned in the previous section.
4. The process is terminated if a utilization goal is reached or hardware restrictions are met.
5. Otherwise the calculation module increases the most over-utilized node (the first node that is utilized more than e.g.: 80%).
6. Goto step 3.

4.2 Real Measurements

Our second approach uses a combination of the previously mentioned calculation module and a measurement approach. Basically, the analytical approach can use hypothetical input data (e.g.: host resources and node service rates) to derive an optimal configuration for the given setup. The goal of our work is to find an optimal solution for a given server software on a specific host. Therefore, another Java module, the measurement module, was developed.

The optimization process of both the calculation and measurement module can be described as follows:

1. The measurement module creates artificial tasks (e.g.: writing data to a file, writing data into a database, data modifications and so forth).
2. These artificial tasks, which should be as close to the tasks of the real queueing network software (e.g.: the DFE) as possible, are assigned to artificial nodes.
3. The measurement module continuously executes the given tasks on a real host and measures the service rate for each node.
4. The measurement module also automatically finds out important hardware specifications of the tested host.
5. The measured service rates, along with the hardware specifications of the tested host are used by the calculation module to recursively find the optimal configuration of the queueing network software for a specific host.

By measuring the service rate on the tested machine, the optimization process now finds the optimal configuration on a given host. In a previously published paper, experiments conducted with the calculation and measurement module are described in more detail in [11]. The focus of this paper, however, is the comparison of the results of our approach with the true optimum configuration for two server machines. These true optimum configurations are derived from experiments conducted with the actual DFE software installed on two server machines (see Section 5).

5 Experiments

To validate our optimization approach a testing environment was set up and experiments were conducted. Therefore, the queueing network software was installed on two hosts (*Goedel* and *Zerberus*).

The first host *Goedel* is a SUN Fire v40z with four dual-core AMD Opteron processors Model 875, each core at 2.2 GHz, has 24 GB of main memory, five 300 GB Ultra320 SCSI HDs, 10/100/1000 Mb/s Ethernet, and runs Linux 2.6.16.60-0.42.7.

The second host *Zerberus* is a Sun SPARC Enterprise T5220, model SED-PCFF1Z with a SPARC V9 architecture (Niagara 2) and a Sun UltraSPARC T2 eight-core processor, each core at 1.2 Ghz and with Chip Multithreading Technology (CMT) for up to 64 simultaneous threads, 32 GB of main memory, two 146 GB Serial Attached SCSI disks, 10/100/1000 Mb/s Ethernet, and runs SunOS 5.10 Generic_127111-11.

Furthermore, an Oracle Database was installed on host *Zerberus*. A ticket generator imitating VoIP devices and sending Diameter tickets to the queueing network software was installed on several test clients.

The procedure of the experiments can be summarized as follows:

- The queueing network software was started with the initial configuration.
- The ticket generator repeatedly sends Diameter tickets to the queueing network software with a given external arrival rate.
- After an experiment cycle, software-internal performance metrics are used to determine the most over-utilized node, which will be increased by one.
- The queueing network software is reconfigured and restarted and a new experiment cycle is started.
- The experiments continue until the optimal configuration is found.

Zerberus. The first experiments were conducted with the actual queueing network software installed on host *Zerberus*. As mentioned before, host *Zerberus* has an Oracle database installed locally and has the possibility to start up to 64 parallel threads.

Again, an over-utilization occurs if the node is utilized more than 80%. Therefore, nodes that show a utilization of over 80% will be increased by one thread.

Table 2 shows the service rate and the utilization of all four nodes with an external arrival rate of 1000 tickets per second. This first experiment makes it obvious that the *Feeder* node is the bottleneck of the system, only managing an average number of 66 tickets per second. Therefore the adaption tool suggests that the *Feeder* node has to be increased by one, creating two *Feeder* threads at the next experiment.

In the next few steps it became obvious that even though the *Feeder* node was recursively extended, the service rate did not increase in the same way. Table 3 shows that the mean service rate of one *Feeder* thread decreases with every newly added *Feeder* thread. Of course, the total service rate of all *Feeder* nodes does not decrease, but adding new *Feeder* threads does not improve the total

Table 2. Initial config. (1-1-1-1) on host *Zerberus* (ext. arrival rate: 1000 tickets/s).

	Service Rate [tickets/s]	Utilization [%]
<i>Decoder</i>	10658	9.38
<i>Converter</i>	12147	8.23
<i>Serializer</i>	2061	48.52
<i>Feeder</i>	66	100.00

service rate of all *Feeder* nodes enough. Even with the maximum number of 61 threads, the *Feeder* node stays the systems bottleneck.

Table 3. Service rates of the *Feeder* node with different configurations on host *Zerberus*

Configuration	Mean Service Rate	
	Individual	Total
(1-1-1-1)	66	66
(1-1-1-2)	44	88
(1-1-1-4)	34	136
(1-1-1-10)	7	70
(1-1-1-20)	5	100
(1-1-1-61)	2	122

Table 4 shows that with the final configuration (one *Decoder*, one *Converter*, one *Serializer* and 61 *Feeder* nodes) all other nodes are of course still under-utilized. This leads to the conclusion that the *Feeder* node should indeed be fixed to one thread per node. The solution to this problem, as mentioned before, could be to allocate a large queue to the *Feeder* node. By doing that, the node can eventually handle queued tickets when the external arrival rate decreases.

Table 4. Final config. (1-1-1-61) on host *Zerberus* (ext. arrival rate: 1000 tickets/s).

	Utilization [%]
<i>Decoder</i>	9.25
<i>Converter</i>	9.67
<i>Serializer</i>	48.95
<i>Feeder</i>	100.00

The final experiment therefore used a *Feeder* node fixed to one thread per node. Table 5 shows that an initial configuration of one thread per each node cannot handle an external arrival rate of 2000 tickets per second without overstepping an utilization of 80%, because the *Serializer* node is already at a

utilization level of 97.04%. Increasing the number of threads per node (without taken the *Feeder* node into account) the final and optimal configuration can handle an external arrival rate of 66900 tickets per second.

Table 5. Initial and final configuration on host *Zerberus*, with a fixed *Feeder* node

Node	Initial Config. 2000 tickets/s		Optimal Config. 66900 tickets/s	
	Util. [%]	Threads	Util. [%]	Threads
<i>Decoder</i>	18.77	1	79.11	9
<i>Converter</i>	16.46	1	70.44	9
<i>Serializer</i>	97.04	1	79.93	45
<i>Feeder</i> (fixed)	100.00	1	100.00	1

Table 5 shows that at an external arrival rate of 66900 tickets per second, all nodes are under a utilization level of 80%. Of course it should be noted, that the *Feeder* node is still highly over-utilized and can only handle about 70 tickets per second. Given the fact that the *Feeder* node and therefore the database is the natural bottleneck, it is necessary to assign a very large queue to the *Feeder* node, to minimize the loss rate.

Goedel. To compare the results and maybe find a different optimal configuration the same experiments were conducted with the queueing network software installed on host *Goedel*. With four dual-core processors, a maximum amount of 8 threads can be started. As mentioned before, host *Goedel* has no local database installed and uses the Oracle database installed on host *Zerberus*.

Table 6 shows the results of the first experiment. At an external arrival rate of 1000 tickets per second, the *Feeder* node is again the systems bottleneck. Table 6 also shows that compared to host *Zerberus*, the *Decoder*, *Converter* and *Serializer* node show a higher service rate during the initial experiment.

Table 6. Initial config. (1-1-1-1) on host *Goedel* (ext. arrival rate: 1000 tickets/s).

	Service Rate [tickets/s]	Utilization [%]
<i>Decoder</i>	32617	3.07
<i>Converter</i>	26058	3.84
<i>Serializer</i>	5202	19.22
<i>Feeder</i>	105	100.00

To start the optimization process, the *Feeder* node again has to be increased. Table 7 shows that on host *Goedel* the individual service rate of one *Feeder* thread

does, to some extent, stay the same, which leads to the fact that the total service rate of all *Feeder* threads is indeed slowly increasing. Table 7 shows that one *Feeder* thread can handle 105 tickets per second, while the final configuration of 5 *Feeder* threads can handle 350 tickets per second.

Table 7. Service rates of the *Feeder* node with different configurations on host *Goedel*

Configuration	Mean Service Rate	
	Individual	Total
(1-1-1-1)	105	105
(1-1-1-2)	79	158
(1-1-1-3)	81	243
(1-1-1-4)	70	280
(1-1-1-5)	70	350

Given the fact that the total service rate of all *Feeder* threads is increasing, it would make sense to stick to this optimization approach. Table 8 therefore shows an initial and optimal configuration on host *Goedel*. With an external arrival rate of 280 tickets per second the *Feeder* node of the initial configuration is over-utilized, but with the optimal configuration of 5 threads, the *Feeder* node is able to stay under the utilization threshold of 80%.

Table 8. Initial and final configuration on host *Goedel*

Node	Initial Config. 280 tickets/s		Optimal Config. 280 tickets/s	
	Util. [%]	Threads	Util. [%]	Threads
<i>Decoder</i>	0.86	1	0.97	1
<i>Converter</i>	1.07	1	1.41	1
<i>Serializer</i>	5.38	1	4.04	1
<i>Feeder</i>	100.00	1	80.00	5

Table 9 shows the results of the experiments, if the software developer decides to fix the *Feeder* node to one thread per node. At an initial configuration of one thread per node and an external arrival rate of 5200 tickets per second, the *Serializer* node would exceed the utilization threshold of 80%. After the optimization process, the system is able to handle up to 15600 tickets per second with the optimal queueing network software configuration (one *Decoder* node, two *Converter* nodes, four *Serializer* nodes and one *Feeder* node), without exceeding the utilization threshold.

Table 9. Initial and final configuration on host *Goedel*, with a fixed *Feeder* node

Node	Initial Config.		Optimal Config.	
	Util. [%]	Threads	Util. [%]	Threads
<i>Decoder</i>	15.94	1	52.10	1
<i>Converter</i>	19.96	1	75.02	2
<i>Serializer</i>	99.96	1	79.89	4
<i>Feeder</i> (fixed)	100.00	1	100.00	1

5.1 Verification of the Analytical Approach

To verify the analytical approach we used the average service rates for each node derived from the experiments done on host *Zerberus* and host *Goedel* (see Table 10) and started the calculation module one more time.

Table 10. Mean service rates of both tested hosts for each node type

Node	Mean Service Rates	
	Zerberus	Goedel
<i>Decoder</i>	9766	30055
<i>Converter</i>	10430	18672
<i>Serializer</i>	1987	6146
<i>Feeder</i>	28	94

Table 11 shows that starting the calculation module with optimization towards throughput, and using the average service rates derived out of the experiments, both, experiments and the calculation module deliver the same optimal configuration. On host *Zerberus* and host *Goedel* a normal optimization would only increase the number of *Feeder* nodes. Due to different hosts and therefore different node service rates, an optimal configuration with a fixed *Feeder* node would lead to an optimal configuration of 9 *Decoder* nodes, 9 *Converter* nodes, 45 *Serializer* nodes and 1 *Feeder* node on host *Zerberus*, and 1 *Decoder* node, 2 *Converter* nodes, 4 *Serializer* nodes and 1 *Feeder* node on host *Goedel*.

These four verifications show the importance of the service rates and if there is no possibility to derive real service rates from an actual software, it is necessary to analyze the used nodes in every detail. As mentioned before, the measurement module is using simulated nodes to derive artificial service rates. Therefore, the simulated tasks have to be as close as they can get to the actual performed tasks of the tested queueing network software.

Table 11. Optimal configuration of threads for both hosts

Optimal Configuration	Zerberus		Goedel	
	Normal	Fixed Feeder	Normal	Fixed Feeder
Experiments	(1-1-1-61)	(9-9-45-1)	(1-1-1-5)	(1-2-4-1)
Analytical Model	(1-1-1-61)	(9-9-45-1)	(1-1-1-5)	(1-2-4-1)

5.2 Removing the Bottleneck

In the above examples we see that the connection to our database is a serious bottleneck that hinders further improvements. Further optimization would therefore try to improve the database connection throughout, e.g., by increasing the network bandwidth, installing new drivers, or installing replicated databases. Consider a hypothetical case where on Zerberus the service rate of the Feeder would be improved by a factor of 5, 10, or even 15.

Table 12. Service rates, number of threads, and utilization in a hypothetical scenario with Feeder being faster by a factor of 5x, 10x, or 15x. The system throughput is increased to 13400, 23000, and 29500 resp.

	5x, EAR: 13400			10x, EAR: 23000			15x, EAR: 29500		
	SR	T	Util. [%]	SR	T	Util. [%]	SR	T	Util. [%]
<i>Decoder</i>	10658	2	62.86	10658	3	71.93	10658	4	69.2
<i>Converter</i>	12147	2	55.16	12147	3	63.12	12147	4	60.71
<i>Serializer</i>	2061	9	72.24	2061	14	79.71	2061	18	79.52
<i>Feeder</i>	330	51	79.62	660	44	79.2	990	38	78.42

Table 12 shows that improving the bottleneck indeed results in a significant improvement of the overall throughput, while keeping all nodes at moderate utilization. Still most threads are invested into the Feeder, which is still the main bottleneck.

6 Conclusion

This work showed how queueing theory can help finding the best configuration of a multi-thread software. By modeling such a software as queueing network consisting of nodes with certain functionalities, optimization towards throughput is possible. As a result the optimal number of threads per node is determined to efficiently use available CPU cores, memory, disk space and speed, and network bandwidth. Experiments evaluated our methodology.

The basic idea behind our three approaches is an online optimization tool that can be placed in front of the queueing network software. After measuring

the respective service times of the nodes, we use an analytical queueing network to find the optimal number of threads for each node.

After that, the data-flow software can go online. The optimization tool should then be able to detect changes in the external arrival rate and – if necessary – recalculate the optimal configuration.

We also conducted several other experiments using a different setup of the queueing network, to see if the proposed approaches can be used for other multi-thread data-flow software. The structure of the queueing network, as well as the used nodes can easily be changed or enhanced, without knowledge of the Java code. Therefore, the mentioned approaches can be used for other multi-thread data-flow software as well.

References

1. Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proceedings of the IEEE 93, 216–231 (2005), <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.136.7045>
2. Püschel, M., Moura, J.M.F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R.W., Rizzolo, N.: SPIRAL: Code Generation for DSP Transforms. Proceedings of the IEEE, Special Issue on Program Generation, Optimization, and Adaptation 93(2), 232–275 (2005)
3. Whaley, R.C., Dongarra, J.J.: Automatically tuned linear algebra software. In: Supercomputing 1998: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing (CDROM), pp. 1–27. IEEE Computer Society, Washington, DC (1998)
4. Osogami, T., Kato, S.: Optimizing system configurations quickly by guessing at the performance. SIGMETRICS Perform. Eval. Rev. 35(1), 145–156 (2007)
5. Balsamo, S., Person, V.D.N., Inverardi, P.: A review on queueing network models with finite capacity queues for software architectures performance prediction. Performance Evaluation 51(2-4), 269–288 (2003)
6. Jain, R.K.: The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling. Wiley, Chichester (1991), <http://www.cse.wustl.edu/~jain/books/perfbook.htm>
7. Zukerman, M.: Introduction to Queueing Theory and Stochastic Teletraffic Models. Zukerman (2009), <http://www.ee.cityu.edu.hk/~zukerman/classnotes.pdf>
8. Bolch, G., Greiner, S., de Meer, H., Trivedi, K.S.: Queueing Networks and Markov Chains: Modeling and Performance Evaluation with Computer Science Applications, 2nd edn. Wiley Blackwell (May 2006), <http://www4.informatik.uni-erlangen.de/QNMC>
9. Kobayashi, H., Mark, B.L.: System Modeling And Analysis - Foundations of System Performance Evaluation, 1st edn., vol. 1. Prentice-Hall, Englewood Cliffs (2008), <http://www.princeton.edu/kobayashi/Book/book.html>
10. Agresti, A.: Categorical Data Analysis, 2nd edn. Wiley-Interscience, Hoboken (2002)
11. Weidlich, R., Nussbaumer, M., Hlavacs, H.: “Optimization towards consolidation or throughput for multi-thread software. In: International Symposium on Parallel Architectures, Algorithms and Programming, pp. 161–168 (2010)