RT-Bench, Improved Understanding of Application Performance with Memory Storage

Zsolt Demeter

Department of Automatic Control

MSc Thesis
TFRT-6043
ISSN 0280-5316

Department of Automatic Control
Lund University
Box 118
SE-221 00 LUND
Sweden

Abstract

By implementing efficient and smart schedulers in our software systems with multiple threads we can make applications run faster and much more efficiently. There is however a lot of caution when adopting and implementing scheduling algorithms, like limited preemptive scheduling or PD^2, due to the uncertainty they may cause on advanced and complex systems. In fact, most algorithms are tested to produce advantages in specific situations. This is one of the reasons why there is a gap between the theoretical scheduling development and the actual schedulers implemented in real operating systems. One way to close the gap is to derive precise guarantees for the implementation of scheduling algorithms, which is the purpose of rt-bench.

Rt-bench calculates characteristic values for a specified scheduling algorithm and a specific task set, mainly in form of supply bound functions based on the execution of the task set on a Linux-based hardware platform. The characteristics can vary depending on the system and the setup, therefore these are used to compare complete execution platforms rather than single algorithms.

This thesis focuses on extending rt-bench to increase the realistic behaviour of the simulation of the application behaviour. Before this thesis, rt-bench could simulate computations but not memory handling. Simulating memory management is necessary to create realistic models and the purpose of this thesis is to introduce this memory usage in rt-bench.

The results show a clear performance drop before the model reaches a memory level equal to the cache size, due to other processes also using the cache memory. This behaviour is what was expected and confirms that the implementation is sufficient for measuring and evaluating performance offered by different platforms.

Aknowledgements

I would like to express my gratitude to my supervisor Assoc. Prof. Martina Maggio for her support and guidance throughout this thesis, for providing me with the support and answers that I needed, and finally for her endless patience when I decided to take a job offer during the thesis, prolonging the process with an unforeseen postponement.

I would also like to thank my fellow student Nicolas for the company and valuable discussions, work related or not, when we were both working late hours with our theses at the university.

Finally I would like to thank my family and friends for believing in me and supporting me when I needed it the most, and thank you Maria for always reminding me that the end was just around the corner.

Contents

1 Introduction

The demand on good resource management techniques, that are quick and fail safe, is high in many areas, especially in areas using real-time thread schedulers. Even though there are many efficient scheduling algorithms available, developers are being very cautious about adopting these, especially in more advanced and complex systems. The reason for this is that most scheduling algorithms tend to be optimal in specific conditions, for example when the load is less than a specific quantity, or to achieve specific goals, for example enforcing fairness. This also makes it hard to choose a scheduler for systems that run multiple applications.

Examples of algorithms that handle specific situations very well can be found in schedulers that operate on mixed criticality system. These systems have two or more levels of criticality, referring to the levels of assurance against application failure (Automotive Safety Integrity Levels, Design Assurance Levels) [Blanquart et al., 2012]. Adding more levels affects scheduling parameters that can completely cripple scheduling algorithms, hence it is important to use flexible schedulers addressing all desired situations [Burns and Davis, 2016].

Several characteristics are relevant in the investigation of scheduling algorithms, such as CPU busyness, number of processed jobs per time unit, response time from task submission and fairness in sharing the CPU between processes [Goel and Garg, 2012]. The overhead from the scheduling algorithms due to schedule and computation context switching needs to be taken in consideration as well [Katcher, Arakawa, and Strosnider, 1993] [Nahas, 2008]. Optimizing all these characteristics is not easy for a single scheduling algorithm and a lot of flexibility would be needed.

Real-time scheduling algorithms usually lack this flexibility and have a high complexity. They are usually optimized to deal with a designated situation. This is one of the reasons why they are usually not implemented in Linux, and more flexible scheduling algorithms are implemented instead. Many scheduling policies are therefore not implemented in Linux or other general purpose operating systems, creating a fundamental gap between the theoretical research on scheduling algorithms and the currently implemented solutions.

Some research has the aim to close the gap between theory and implementation. An example of an application trying to do this is LITMUS^{RT} [Calandrino, Leontyev, Block, Devi, and Anderson, 2006]. LITMUS^{RT} is simplifying the the implementation of new schedulers to Linux platforms, therefore making it easier to test a new scheduling policy.

Another approach to close the gap is determining real-time characteristics of existing implementations. This is being done by extracting complex guarantees on scheduling algorithms when they are run. This approach is being adopted by the application used in this thesis, called rt-bench [Maggio, Bini, and Lelli, 2016].

This thesis extends rt-bench. Before this thesis began, rt-bench could be used to simulate the behaviour of applications doing computations and taking and releasing locks on shared resources. During the course of this thesis, rt-bench was improved introducing memory allocation and usage, allowing the user to create more realistic application models in rt-bench.

This introduction will start by explaining the goal and aim of this thesis, followed by the given constraints and a thesis outline.

1.1 Problem Statement

This thesis aims to improve the analysis capabilities of rt-bench by implementing memory usage. The problem statement and questions before the start of the thesis were:

- How does the amount of memory used interfere with the computing performance delivered by the platform?
- How will the application memory consumption affect the real-time execution characteristics?
- Can the same real-time guarantee be provided in case of memory usage?

1.2 Goal

The goal of this thesis is to introduce more accurate and realistic application models in rt-bench. The result of that would be a better understanding of the behaviour of specific applications in certain environments and with specific conditions. This would have a positive impact in the field of resource management techniques and operating system developers would be able to select efficient execution environments with the confidence of maintaining the real-time guarantees.

By implementing memory usage in the simulated applications and letting an application use not only computation power, but also memory, a change in the timing results is expected. From this information a relation between the amount of memory used and the provided real-time guarantees during application scheduling will be obtained.

To predict application behaviour to different changes and implementations is not trivial, which makes it important to evaluate the changes and implementations that are being made to an application if it has to deliver real-time performance guarantees.

1.3 Constraints

When an operating system is executing, the results will vary depending on several factors. Examples of these factor are:

- thermal stress,
- frequency scaling,
- interrupts from external devices.

This makes it hard to predict the result from simulations and the results will differ depending on the equipment quality, thus making this application more suitable for comparing schedulers among each other [Maggio, Bini, and Lelli, 2016].

1.4 Thesis outline

The structure of this thesis is designed to improve readability. In the first chapters, the thesis presents all the information needed to fully understand the extent of the work and the results obtained. In particular, Chapter 2 describes the starting point for this work and Chapter 3 presents the theoretical background needed for the results interpretation.

These chapters are followed by Chapter 4, where the contribution to rt-bench is described and Chapter 5, that covers the work process. The collected information will be presented and discussed in the Chapters 6 and 7.

2

Theoretical Background

This chapter provides an overview of the theory needed to understand the results of the tests performed with rt-bench. It starts with general information about the platform being used and continues by explaining the numerical background.

2.1 Scheduling in Linux Kernels

To run multiple processes, the Linux kernel uses an algorithm to decide the order and duration of the processes' execution. The algorithm must fulfill several requirements, for example, a good response time, or an appropriate amount of processing capacity to background jobs, or to avoid process starvation. This algorithm is usually composed of a set of rules, and is often referred to as a *scheduling policy* [Bovet and Cesati, 2005].

Time Quantum

Scheduling in the Linux kernel is based on a time-sharing technique with a multiplexing concept. The result of this technique can be visualized as the CPU time between applications being divided into *slices*. A single processor can only run one process at a given instant, so each processor is alternating between the active processes [Bovet and Cesati, 2005].

Some scheduling algorithms use quanta to limit these time slices and for dynamic scheduling. If the quantum of the current running process expires, a context switch may take place if another process is waiting for the CPU.

The length of a time slice or a quantum is a critical parameter for system performance since different lengths affect the performance of the running infrastructure. If the quantum is small, context switches will occur often, introducing high additional overhead. If the quantum is large the user experience will suffer and the user will not have the impression that different tasks are executed in parallel [Bovet and Cesati, 2005].

Process Priority

To determine the order in which the processes will be run, in Linux each process is associated with a *scheduling policy* and a static scheduling priority, *sched_priority*. The latter is only used for scheduling decision for real-time processes. For other processes the priority is set to 0, and scheduled depending on their *nice value*. Processes scheduled with a real-time policy will have a priority value in the range of 1 to 100 where 1 is low priority and 100 is high [Kerrisk, Zijlstra, and Lelli, 2016a].

Each CPU core for a Linux platform has a priority tree with 100 *runqueues* to keep track of the priority of all processes. The processes are put in the queue with the associated priority. The scheduler then runs the process at the head of the highest nonempty queue. This means that a real-time process will always run before a normal one. Within the queues the run order is determined by a *dynamic priority* set by the scheduling policy associated with the process. A scheduler can be preemptive or non-preemptive. In schedulers that are preemptive, processes with higher priority are allowed to interrupt processes with lower priority in order to run. All schedulers in Linux are preemptive, meaning all scheduling in the runqueue is also preemptive, and if a process is interrupted it will be returned to the appropriate runqueue depending on its priority [Kerrisk, Zijlstra, and Lelli, 2016a].

As mentioned, there are both normal and real-time scheduling policies available in the Linux kernel. The available normal policies are:

- `SCHED_OTHER` is the default Linux scheduler and increases its dynamic priority each time quantum the process is ready to run, but denied by the scheduler. It uses the standard scheduling algorithm in Linux, called the *Completely Fair Scheduler*.
- `SCHED_IDLE` (introduced in version 2.6.23 of the Linux kernel) is used for jobs with extremely low priority.
- `SCHED_BATCH` (introduced in 2.6.16) is similar to `SCHED_OTHER` but it will always tell the scheduler that a process is CPU-intensive for it to get a small scheduling penalty with respect to wake-up behaviour.

The available real-time policies are:

- SCHED_FIFO will run a process until it gets blocked, preempted or calls kernel command sched_yield(2), stopping itself. The process will be placed in the back of the list for its priority when it becomes runnable, although if preempted it will stay at the head.
- SCHED_RR works like SCHED_FIFO except each process is only allowed to run for a set amount of time before being put at the end of the list. The time that passes, if the process gets blocked or preempted, is not included.
- SCHED_DEADLINE (introduced in 3.14) is implemented using *Global Earliest Deadline First* and *Constant Bandwith Server*. This policy has the highest priority amongst all policies. When a process is run with this policy it will always preempt a thread running with one of the other policies. This would correspond a process with priority 100. [Kerrisk, Zijlstra, and Lelli, 2016a]

When multiple cores are available, there is a runqueue for each core. An entering process is put in one of these runqueues by the Linux scheduler. To keep the efficiency high, a logical global runqueue is implemented. This runqueue fires push and pull operations to balance the load on the cores. If the head of a runqueue is modified, a push operation will be executed to see if the new process at the head can be pushed to another runqueue. If instead, a process suspends itself or lowers its priority, a pull operation will be executed and check if there are processes on the other cores with higher priority that can be migrated to this core [Lelli, Lipari, Faggioli, and Cucinotta, 2011].

A process can also be restricted to specified cores by setting the *affinity* property. In this case the push an pull operations will not affect this process [Kerrisk, Zijlstra, and Lelli, 2016b].

Process classification

Processes, or threads, can be classified as I/O-bound (input/output-bound) or CPU-bound. I/O-bound processes use communication devices or peripherals, such as mouse and keyboard, and spend a lot of time waiting for I/O operations. On the contrary, processes that are CPU-bound perform computations and require CPU time. The processes can be divided into three other classes that are relevant during process scheduling. These are *interactive processes*, *batch processes* and *real-time processes*.

Interactive processes are processes that interact constantly with users, and spend a lot of time waiting for user operations. Because of the interaction with users, the average delay of these processes must be low and have a short variance. If these requirements are not satisfied the processes will be found unresponsive.

The *batch processes* do not interact with users and are often run in the background. There are no strict requirements for these and they are therefore often penalized by the scheduler.

The *real-time processes* have very strict timing requirements due to expectations on response times and deadlines. The response time should be deterministic with a minimum variance or disruptive consequences may occur [Bovet and Cesati, 2005].

2.2 The Supply Function

Supply functions are functions that capture some characteristics of scheduling platforms. These characteristics are dependant on the scheduling policy, the given task set and the hardware. The supply functions try to express the computing capacity offered to a process, and as a *supply lower bound functions* and a *supply upper bound function* can be computed, upper and lower bounds of the computing capacity offered will also be expressed [Mok, Feng, and Chen, 2001].

The operations of the scheduler can be modeled by a *scheduling function $s_i(t)$* and described as

$$s_i(t) = \begin{cases} 1 & \tau_i \text{ runs at } t \\ 0 & \text{otherwise.} \end{cases} \tag{2.1}$$

The overall schedule $s_*(t)$ over the platform $\mathscr{P}$ is defined as

$$s_*(t) = \sum_{\tau_i \in \mathscr{T}} s_i(t). \tag{2.2}$$

An example of a scheduling function can be seen in Figure 2.1 and is represented by Equation 2.2.

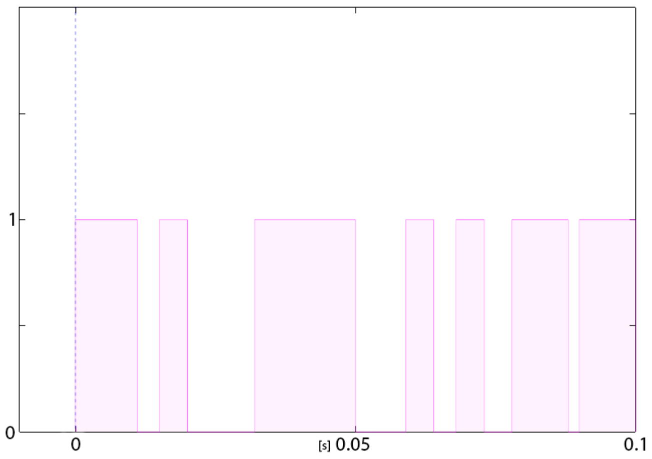


Figure 2.1: Example of a scheduling function where the process is occasionally getting CPU power.

The overall schedule is limited by $\mathscr{P}$ such that only a certain number of threads can run in parallel limited by the amount of available CPUs. If $\mathscr{P}$ allows at most m threads to run in parallel then $s_*(t) \leq m$. An abstraction of execution platforms can be given by a *supply function*.

Using 2.1 and 2.2 we can define two more functions called the *supply lower bound function* ($slbf(t)$) and the *supply upper bound function* ($subf(t)$) as presented below:

$$slbf(t) = \min \int_{t_0}^{t_0+t} s(x)dx. \tag{2.3}$$

$$subf(t) = \max \int_{t_0}^{t_0+t} s(x)dx. \tag{2.4}$$

An illustration of how the supply lower bound function and upper bound function are defined can be made from the scheduling function.

By calculating the integral of the scheduling function, a curve similar to a supply function is obtained. The difference, when calculating the supply lower and upper bound functions, is that the time interval being integrated is placed dynamically on the x-axis to minimize or maximize the integral values.

Starting with the supply upper bound function, the Δt will be computed dynamically on the schedule axis so that the total time when the process receives CPU is maximized. As Δt is increased it will be placed optimally to maximize the integrated area which is illustrated in Figure 2.2.

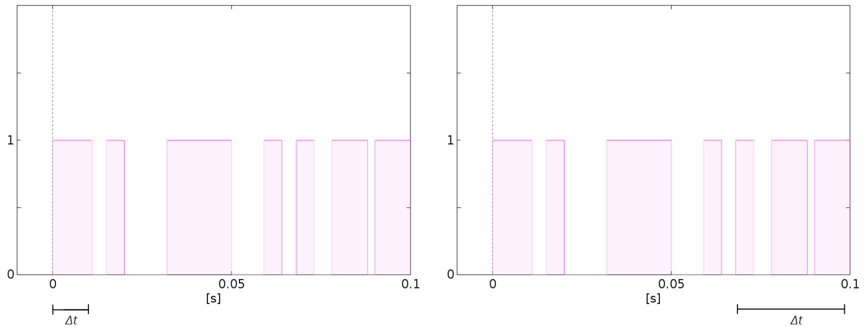


Figure 2.2: Δt being placed where the total CPU time is maximized, depending on the size of Δt.

The same derivation is applied to the lower bound function, but instead calculated with the minimum integrated area, as seen in Figure 2.3. The resulting curves can be seen in Figure 2.4.

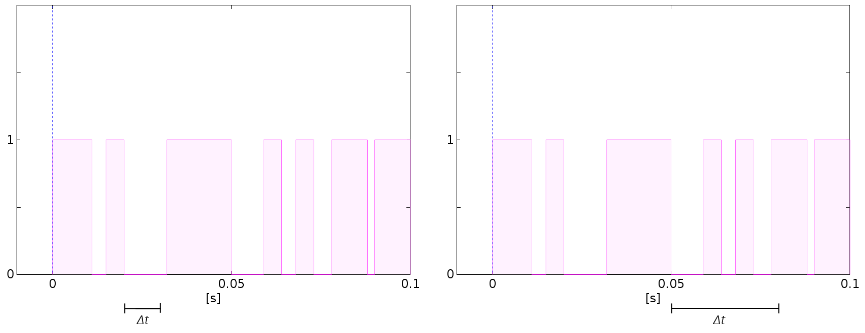


Figure 2.3: Δt being placed where the total CPU time is minimized, depending on the size of Δt.

In summary, the x-axis shows a time interval, Δt, and the y-axis shows the total amount of processing capacity that is being received in seconds for the specific time interval. An example of a supply lower bound function and a supply upper bound function is illustrated in Figure 2.4.

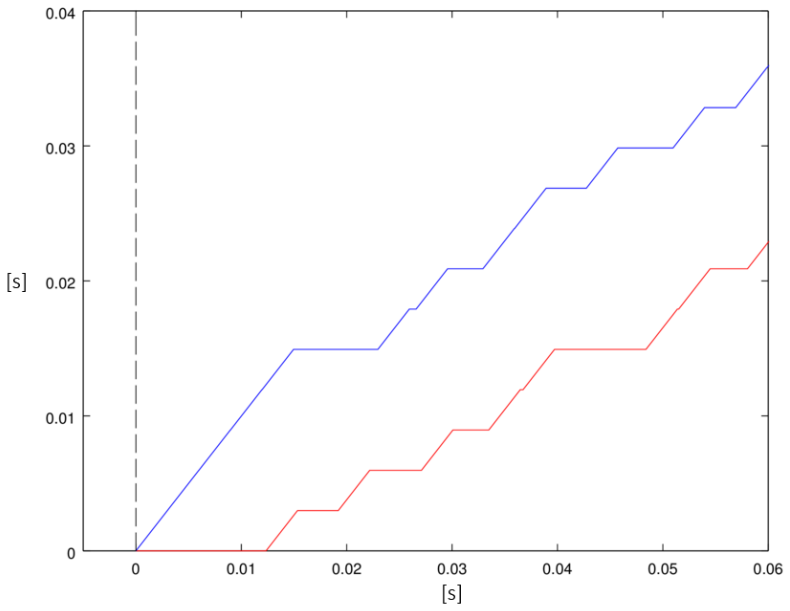


Figure 2.4: Example of lower and upper bound supply functions

The supply lower and upper bound functions from a single test does not show the exact behaviour of it. Instead these indicate a range for how the actual process has behaved meaning that if the functions are close to each other the interval between the two functions represent a good approximation of the resources given to the task.

3

Implementation Background: rt-bench

This chapter introduces rt-bench, which is used to perform experiments on a specific task set using some scheduling algorithm and on a specific architecture. The data collected during these experiments is then analyzed to provide some insight on the schedulers' behavior. The chapter includes a description of the status of rt-bench prior to this thesis, while the specific contribution of this work is then described in Chapter 4.

The rt-bench application intends to investigate if real-time guarantees are provided from existing algorithm implementations by determining their real-time characteristics, mainly a characteristic called *supply function*. The investigation takes place when rt-bench executes a model created of a system or an application. To retrieve valuable results it is of great importance that realistic application models are being used thus the possibility to create realistic models needs to be implemented and available.

When rt-bench executes, several actions are performed. Rt-bench starts with running an experiment for a set amount of time, with a specified application, on a chosen platform, and with a chosen scheduler. During the process, timestamps are extracted at critical moments and analyzed to determine the real-time behaviour of the run.

Rt-bench analyzes the supply functions described in Section *2.2*. The results given from the supply functions vary with the amount of CPU that is offered to the threads during the tests and that in turn depends on the machine, operating system, choice of scheduling algorithm, and application characteristics.

3.1 Platform specifications

The experiments are executed on two computers with a Linux operating system, one is a launcher and one is a target machine. The launcher's purposes are:

1. to tell the target what tests to execute,
2. to analyze the logs.

This results in the target machine not being affected by software and memory usage on the launcher. This lets the launcher contain scripts to run multiple tests, stored data, etc. An illustration of the setup can be seen in Figure 3.1.

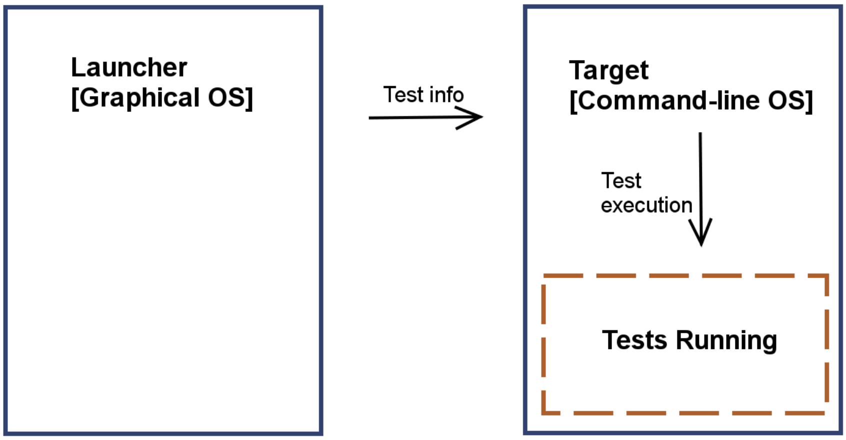


Figure 3.1: Execution flow for running the rt-bench application.

Because of this the only relevant specification needed is the one of the target machine, which follows:

- Intel(R) Core(TM)2 Duo Processor
- Clock speed at 3.00GHz

- Cache size of 6144 kB
- Disk space, 648 GB
- 32-bit Ubuntu 14.04 LTS

3.2 Rt-bench

Rt-bench's task is to monitor the execution of a synthetic application regulated by a chosen scheduling algorithm and executed on a top of a specific architecture. This application is composed of a set of n threads specified by the user itself and denoted with $\mathscr{T} = \{\tau_1, ..., \tau_n\}$. The threads have a *job body* that is executed in a loop until the user-defined test duration is over. The *job bodies* of thread i consist of p_i sequentially defined *job phases*. The *phases* are denoted by $\phi_{i,1}$,$\phi_{i,2}$,...,ϕ_{i,p_i} and each of them belongs to a set of available *phases*, $\phi_{i,p_i} \in \Phi$. The available phases are listed and described in Section 3.4.

In an iteration of a *job body*, the entire sequence of *job phases* is executed. Each time before evey job begins an event will be fired. To record these events an external timestamping tool called *trace_cmd* [*TRACE-CMD(1), man page* 2015], is used. The test flow can be seen in Figure 3.2.

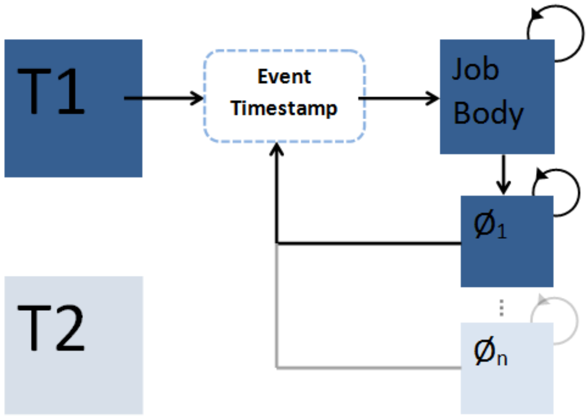


Figure 3.2: Test flow and time stamp initiation.

To define and run these threads, the user has to create a *json test configuration file*. The json-file contains information about all the threads that are going to be executed by the rt-bench application. The exact structure and information of this file is described in Section 4.2. The threads will be executed over a platform $\mathscr{P}$ which is characterized by a computing capacity, a scheduler and an operating system.

3.3 Alpha and Delta Characteristics

Rt-bench generates empirical supply functions based on the execution of the threads during the experiments and extrapolates two parameters, α and Δ. Other names that are used for these are the *bandwidth* and the *delay*. Thes e two values are defined in pairs and are connected with a supply lower bound function $slbf(t)$ or supply upper bound function $subf(t)$ if

$$\forall t, \quad slbf(t) \leq \alpha_{lower}(t - \Delta_{lower}) \tag{3.1}$$

$$\forall t, \quad subf(t) \geq \alpha_{upper}(t - \Delta_{upper}). \tag{3.2}$$

With an obtained α and Δ it is possible to approximate and interpret the supply functions with a line and a slope instead of the generated "staircase" function. This will make it easier to relate to the variables, thus creating a greater understanding for the results.
Using the definition in [Buttazzo, 2011], the bandwidth α for any of the given supply functions can be defined as the asymptotic bandwidth in an arbitrarily large interval for t with

$$\alpha_{lower} = \lim_{t \to \infty} \frac{slbf(t)}{t} \tag{3.3}$$

$$\alpha_{upper} = \lim_{t \to \infty} \frac{subf(t)}{t} \tag{3.4}$$

while Δ is defined as

$$\Delta_{lower} = \sup_{t \geq 0} \left\{ t - \frac{slbf(t)}{\alpha_{lower}} \right\} \tag{3.5}$$

$$\Delta_{upper} = \inf_{t \geq 0} \left\{ t - \frac{subf(t)}{\alpha_{upper}} \right\}. \tag{3.6}$$

For our experiments the time intervals will not be arbitrarily large since that is only theoretically possible. The intervals will be limited, but to adapt to the experiment length another definition has been adopted. Figure 3.3 shows an example of the adapted definition.

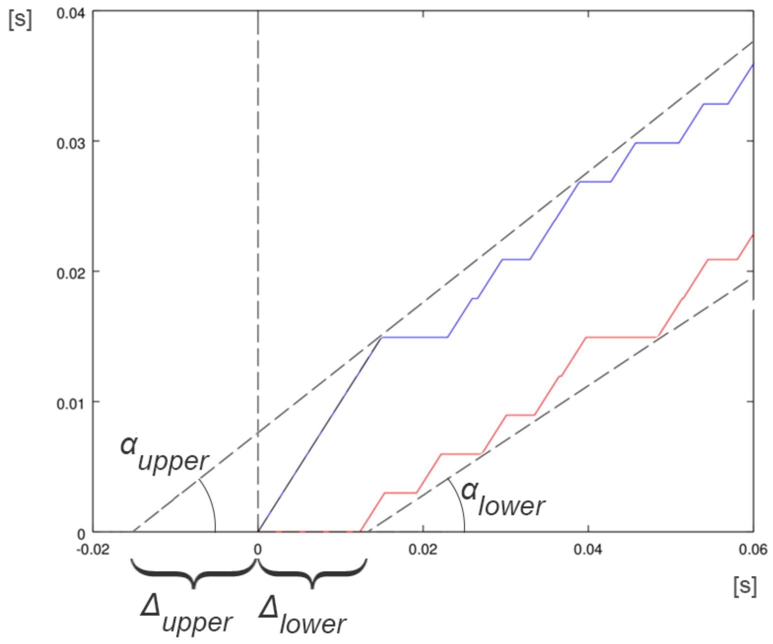


Figure 3.3: Adapted interpretation of Alpha(α) and Delta(Δ).

3.4 Job Phases

The job phases are created separately so that each phase is represented by a function in the code. The reason for this is easier reproduction of application code. When using rt-bench a real application will be in focus, and from this application it is desired to extract a model to be used. Depending on what the application does the correct phases should be used to create a model that resembles the real application [Maggio, Bini, and Lelli, 2016, sec. IV-A]. The job phases are further customized by the user by having input arguments that decides how the simulated application will behave during execution. The two available phases when this thesis started were the *compute phase* and the *lock phase*.

Compute Phase

The compute phase covers the most basic purpose of an application by doing computations. The computation is simulated by executing mathematical operations and using the CPU in the lightest way. The computation consist of addition, subtraction and library calls. The data produced is not being used anywhere other than temporarily being saved on the stack as a local variable.

All applications use computing similar to this when manipulating data and this phase is used to simulate the parts of a process where this is the only thing being performed. Some examples that partly need the compute phase to be simulated are image and sound processing and execution of a controller.

The input argument for the compute phase is an amount of operations to be made in the job phase before it is considered done.

Lock Phase

The lock phase comes in use when there are applications using multiple threads that have shared resources. Besides doing the same computations as the compute phase it also acquires a resource, blocking other threads that are waiting for the same resource.

The lock phase requires two arguments where the first one is the amount of operations to be completed in the critical section, just like in the compute phase, and the second one is a *resource id*. The resource id specifies what resource should be acquired by the thread. Threads can depend on different resources which makes the lock phase open for further customization.

4

Thesis Contribution

This chapter presents the contributory work that is aiming to make the application model used in rt-bench more realistic. The contribution consists in adding an additional phase called *memory phase*. The new set of available phases then becomes $\Phi = \{\phi^{compute}, \phi^{lock}, \phi^{memory}\}$.

The configuration of the phases using json-objects will be described along with mentioning the automation that was used.

4.1 Memory Usage Phase

The aim of this thesis is to enable simulations with memory storage. To do so the memory phase is added to the set of available job phases Φ. It is not unusual that an application would want to store and use data on the system and with the memory phase it will be possible to execute more realistic models of these applications.

Whenever something is saved on a computer an action similar to the memory phase will execute. Some simple examples are a text editor saving a document, or a recording application for a video camera or microphone.

The memory phase takes two inputs when being executed and for convenience the second argument will be explained first. The second argument is the amount of the data type double that should be saved. The size of a double is 8 bytes, meaning that the second argument represents the amount of sets with 8 bytes that should be stored. The first argument that the memory phase requires is the amount of operations, or times the amount of memory specified with the second argument should be stored. The code for the memory phase can be seen below.

```
void memory (int ind, ...) {
  int memory_used, loops, i;
  double *accumulator;
  struct timespec *t_spec;
  va_list argp;
  va_start(argp, ind);
  t_spec = va_arg(argp, struct timespec*);
  memory_used = va_arg(argp, int);
  va_end(argp);
  loops = timespec_to_usec(t_spec);

  accumulator=(double*)malloc(memory_used*
     sizeof(double));
  for (i = 0; i < loops; i++) {
    accumulator[i%memory_used]+=0.5;
    accumulator[i%memory_used]-=floor(
       accumulator[i%memory_used]);
  }
  free(accumulator);
}
```

The code show that the function *malloc* is being used. This means that the data allocated will be stored on the heap. The heap is a dynamic memory storage unlike the stack that only stores temporary variables only accessed by the local function [Shaw, 2015].

Since the purpose of this function is to simulate the usage of memory, the purpose would be lost if the compiler optimized away the operations. In order to prevent any unwanted compiler optimization the stored values need to be used. By using the previous value in the assigning operation (with the $+=$ operator) the compiler interpret it as being used and execute the operation. Additionally, to prevent unwanted overflows, the assigned value is reduced with the $floor()$ function. This will keep the stored values at either 0 or $0,5$.

4.2 .json-file Configuration

The JSON format is a text format that simplifies data exchange between different programming languages. The word is short for *JavaScript Object Notation*. By maintaining a simple file structure all languages will be able to relate to the information [*The JSON Data Interchange Format* 2013]. In rt-bench the JSON-structure is used to parse and store experiment configurations.

```
{
  "resources": 1,
  "tasks": {
    "thread1": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "m0": { "loops": 10000, "memory": 1000000 }
        "m1": { "loops": 10000, "memory": 5000000 }
      }
    },
    "thread2": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "c0": { "loops": 150000 }
        "l0": { "loops": 100000, "res": 0 }
      }
    }
  },
  "global": {
    "default_policy": "SCHED_OTHER",
    "duration": 50,
    "logdir": "./",
    "logbasename": "rtbench",
    "lock_pages": true,
    "ftrace": true
  }
}
```

Example of a json-file used in rt-bench.

This example of a json-file consist of two threads called *thread1* and *thread2*. Each of these threads has a job body and the phases that are to be executed are inside the *phases* section. The type of phase depends on the name that is given to it and the names of the phases here are *m0, m1, c0, l0*. Any name that starts with an *m* corresponds to a memory phase, any name starts with a *c* corresponds to a compute phase, and finally any name that starts with an *l* corresponds to a lock phase.

The part named *global* in the example contains some information designing the test, for example the scheduler to be used and test duration. The duration time decides how long the application will run meaning that each job body will keep rerunning until this time is over and it is after these reruns that events are being fired.

The *default_policy* is set to `SCHED_OTHER`. This is the input that decides what scheduling algorithm is being used during the test. As in the example, all the tests run in this thesis will be using the `SCHED_OTHER` option. This algorithm is the standard linux time-sharing scheduler and has a dynamic priority which means that the priority is set first inside the list [Kerrisk, Zijlstra, and Lelli, 2016a].

In this example the first thread is populated with two memory phases and the second thread is populated with a compute phase and a lock phase. So what happens in thread1 is that first the *m0* phase allocates 1000000 doubles and uses them 10000 times. When this is finished the second memory phase will allocate 5000000 doubles 10000 times. When the second phase has finished it starts over.

Simultaneously thread2 will run it's phases. First it will run a compute phase *c0* that performs 150000 operations and then it will run a lock phase *l0* that will perform 100000 operations while locking a resource with resource id *0*. When this is done the thread will start over. This will continue like this for 50 seconds, the amount of time set as the *duration*.

4.3 Automating test generation and result collection

For the results contained in this thesis, several hundred tests were generated, therefore manually executing these tests as described above would be nearly impossible. Several scripts have been made to consecutively run all the tests in an effective way and gather the data. Some examples of this is the shell script *xlaunch.sh* that runs a set of pre-specified set of tests automatically, and *make_outputs* that collects and summarize the significant data from certain sets. This however is nothing that affects the test results, but rather enables the tests to be quantified. Documentation of all added scripts can be found in Appendix A.

5

Test Process

To execute rt-bench a shell script called *launch.sh* is used. This script require some specific parameters to consecutively launch all scripts and tools to complete the tests. The command line to do this in general is:

$$\$./launch.sh =< Ip >< Port >< Uname >< json >< Tname >$$

Where $< Ip >$ is the IP-adress to the target machine that will execute the tests, $< Port >$ is the port, $< Uname >$ is the target machines user name, $< json >$ is the patch to the desired .json file to be executed and $< Tname >$ is the userdefined name of the test.

5.1 Tests

The purpose of all the tests is to see how the usage of the newly implemented memory phase affects the results and if the results have become more realistic. The focus lies in comparing the upper and lower supply functions with and without memory usage. In most tests the amount of memory used is being varied to see the impact of this.

The tests are alphabetically ordered and are named after forenames. Tests which have names starting with the same letter are connected to each other with the purpose of comparison.

The Albert tests are intended as an initial test suite where one thread runs alone to get a base behaviour of the memory usage.

As opposed to the Albert tests, the Bart, Bert and Burt tests will have an additional thread running to see how this would interfere with the measured thread.

The Casper and Ceasar tests are run to see how the initialization and termination of other threads effect the measured thread.

Finally the Dumble tests have the intention of giving a visualization of the overall behaviour of two threads where both have memory allocation.

All tests have the intention of providing proof for the memory phase being implemented in a correct way, and behaving in a predictable and realistic way.

Albert Tests

The following JSON data shows the characteristics of the Albert tests.

```
{
  "resources": 0,
  "tasks": {
    "thread1": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "m0": { "loops": 100000, "memory": X }
      }
    }
  },
  "global": {
    "default_policy": "SCHED_OTHER",
    "duration": 50,
    "logdir": "./",
    "logbasename": "rtbench",
    "lock_pages": true ,
    "ftrace": true
  }
}
```

The scheduling policy will be set by the SCHED_OTHER option and the duration for this test will be 50 seconds.

The thread used in this test have access to one CPU and consist of one memory phase. This memory phase consist of 100.000

executions and the amount of memory X is being varied and $X \in [10, 100.000.000]$ with a logarithmic inclination (10, 20, ..,90 ,100, 200, ...,900,1000,2000, ...). The value of the varied amount of memory is inserted at the red X in the JSON data for each test. The total amount of values in $[10, 100.000.000]$ with the logarithmic increase is 64, resulting in the Albert Test series consisting of a total of 64 tests.

Bart Tests

The following JSON data shows the characteristics of the Bart tests.

```
{
  "resources": 0,
  "tasks": {
    "thread1": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "m0": { "loops": 100000, "memory": X }
      }
    },
    "thread2": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "c0": { "loops": 100000 }
      }
    }
  },
  "global": {
    "default_policy": "SCHED_OTHER",
    "duration": 50,
    "logdir": "./",
    "logbasename": "rtbench",
    "lock_pages": true,
    "ftrace": true
  }
}
```

The scheduling policy will be set by the SCHED_OTHER option and the duration for this test will be 50 seconds.

Thread1 in this test have access to one CPU and consists of one memory phase. This memory phase consists of 100.000 executions and the amount of memory X is being varied and $X \in [10, 100.000.000]$ with a logarithmic inclination.

Thread2 also have access to one CPU and consists of one compute phase. This compute phase is kept static consisting of 100.000 executions. The Bart Tests consist of a total of 64 tests.

Bert Tests

The following JSON data shows the characteristics of the Bert tests.

```
{
  "resources": 0,
  "tasks": {
    "thread1": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "m0": { "loops": 100000, "memory": X }
      }
    },
    "thread2": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "m1": { "loops": 100000, "memory":
            1000000 }
      }
    }
  },
  "global": {
    "default_policy": "SCHED_OTHER",
    "duration": 50,
    "logdir": "./",
    "logbasename": "rtbench",
    "lock_pages": true ,
    "ftrace": true
  }
}
```

The scheduling policy will be set by the SCHED_OTHER option and the duration for this test will be 50 seconds.

Thread1 in this test have access to one CPU and consists of one memory phase. This memory phase consists of 100.000 executions and the amount of memory X is being varied and $X \in [10, 100.000.000]$ with a logarithmic inclination.

Thread2 also have access to one CPU and consists of one memory phase. This memory phase is kept static, consisting of 100.000 executions and have a memory usage of 1.000.000 doubles. The Bert Test series consist of a total of 64 tests.

Burt Tests

The following JSON data shows the characteristics of the Burt tests.

```
{
  "resources": 0,
  "tasks": {
    "thread1": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "m0": { "loops": 100000, "memory": X }
      }
    },
    "thread2": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "m1": { "loops": 100000, "memory": X }
      }
    }
  },
  "global": {
    "default_policy": "SCHED_OTHER",
    "duration": 50,
    "logdir": "./",
    "logbasename": "rtbench",
    "lock_pages": true,
    "ftrace": true
  }
}
```

The scheduling policy will be set by the SCHED_OTHER option and the duration for this test will be 50 seconds.

Thread1 in this test have access to one CPU and consists of one memory phase. This memory phase consists of 100.000 executions and the amount of memory X is being varied and $X \in [10, 100.000.000]$ with a logarithmic inclination.

Thread2 is identical to *thread1* in such way that the memory used in the tests will always be same. The Burt Test series consist of a total of 64 tests.

Casper Tests

The following JSON data shows the characteristics of the Casper tests.

```
{
  "resources": 0,
  "tasks": {
    "thread1": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "m0": { "loops": 100000, "memory":
            1000000 }
      }
    },
    "thread2": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "c0": { "loops": Y }
      }
    }
  },
  "global": {
    "default_policy": "SCHED_OTHER",
    "duration": 50,
    "logdir": "./",
    "logbasename": "rtbench",
    "lock_pages": true,
    "ftrace": true
  }
}
```

The scheduling policy will be set by the SCHED_OTHER option and the duration for this test will be 50 seconds.

Thread1 in this test have access to one CPU and consists of one memory phase. This memory phase is kept static, consisting of 100.000 executions and have a memory usage of 1.000.000 doubles.

Thread2 also have access to one CPU but consists of one compute phase. The amount of executions Y for this thread is being varied and $Y \in [3000, 300000]$ with a linear inclination. The Casper Test series consist of 100 tests.

Ceasar Tests

The following JSON data shows the characteristics of the Ceasar tests.

```
{
  "resources": 0,
  "tasks": {
    "thread1": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "m0": { "loops": 100000, "memory":
            1000000 }
      }
    },
    "thread2": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "c0": { "loops": Y }
      }
    },
    "thread3": {
      "priority": 10,
      "cpus": [0],
      "phases": {
"c1": { "loops": Y }
      }
    },
    "thread4": {
      "priority": 10,
      "cpus": [0],
      "phases": {
"c2": {"loops": Y }
      }
    }
  },
  "global": {
    "default_policy": "SCHED_OTHER",
    "duration": 50,
    "logdir": "./",
    "logbasename": "rtbench",
    "lock_pages": true,
    "ftrace": true
  }
}
```

The scheduling policy will be set by the SCHED_OTHER option and the duration for this test will be 50 seconds.

Thread1 in this test have access to one CPU and consists of one memory phase. This memory phase is being kept static consisting 100.000 executions and have a memory usage of 1.000.000 doubles.

Thread2 also have access to one CPU but consists of one compute phase. The amount of executions Y for this thread is being varied and $Y \in [1000, 100.000]$ with a linear inclination.

Thread3 and *thread4* are identical to *thread2*. The Ceasar Test series consist of 100 tests.

Dumble Tests

The following JSON data shows the characteristics of the Dumble tests.

```
{
  "resources": 0,
  "tasks": {
    "thread1": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "m0": { "loops": 100000, "memory": X1 }
      }
    },
    "thread2": {
      "priority": 10,
      "cpus": [0],
      "phases": {
        "m1": { "loops": 100000, "memory": X2 }
      }
    }
  },
  "global": {
    "default_policy": "SCHED_OTHER",
    "duration": 50,
    "logdir": "./",
    "logbasename": "rtbench",
    "lock_pages": true,
    "ftrace": true
  }
}
```

The scheduling policy will be set by the SCHED_OTHER option and the duration for this test will be 50 seconds.

Thread1 in this test have access to one CPU and consists of one memory phase. This memory phase consists of 100.000 executions and the amount of memory X_1.

Thread2 also have access to one CPU and consists of one memory phase. This memory phase consists of 100.000 executions and the amount of memory X_2.

The amount of memory X_1 and X_2 is being varied seperately and $(X_1, X_2) \in [10000, 1000000]$ with a logarithmical inclination in *thread1* and *thread2*. The Dumble Test series consist of 361 tests.

6

Results and Discussion

This chapter contains plots from all tests that has been executed with rt-bench.

The desired results from rt-bench is to retrieve the behaviour of the created model thus getting an estimated behaviour of the actual application. Since rt-bench produces a lower and upper supply function it is only known that the model behaviour is somewhere between these. This makes it crucial that the lower and upper supply functions do not diverge too much. As long as the behaviour can be limited inside a small enough gap it will be possible to make safe conclusions about algorithm and process behaviour.

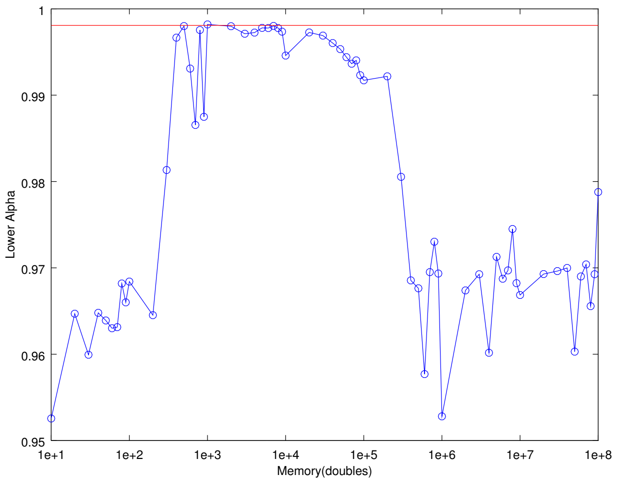
The consistency of the supply functions are easiest visualized with the α characteristics. As long as the supply lower and upper bound functions do not diverge too fast, the behaviour of the real supply function will be somewhat known. If the α values of the supply functions are close to each other the curves will not diverge rapidly.

If the α values differ a lot, meaning the lower and upper supply functions are diverging fast, the estimation on the actual resource provided by the platform becomes unreliable. The edges of the upper and lower supply functions would spread forming a cone and the range of possible positions of the actual supply function would be too wide. If the Δ-difference was the only characteristic that was large, the theoretical supply function would most likely still be within a reasonable range, but as for two diverging curves the range would go towards infinity.

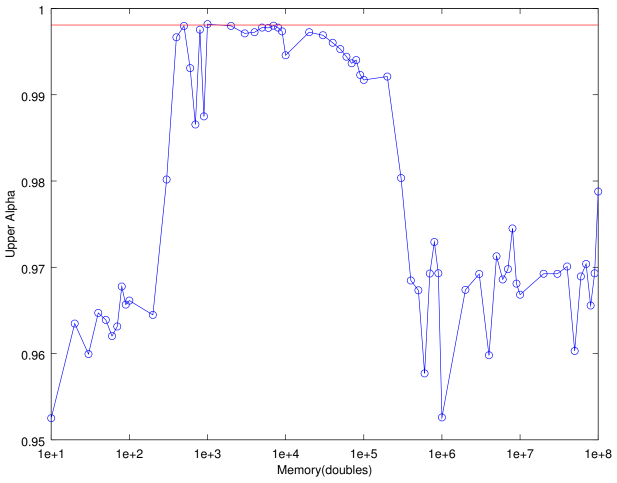
6.1 Albert Tests

The purpose of the Albert tests were to establish a base behaviour, and to see that the addition of a memory phase would reflect the results. An additional red line has been added to the plots to visualize what the α and Δ values would have been if the memory phase would not have been present.

In these initial tests the α values were expected to be lower than the red line. With the addition of the memory handling, the thread is expected to be given less run time and give a lower slope to the α illustration.

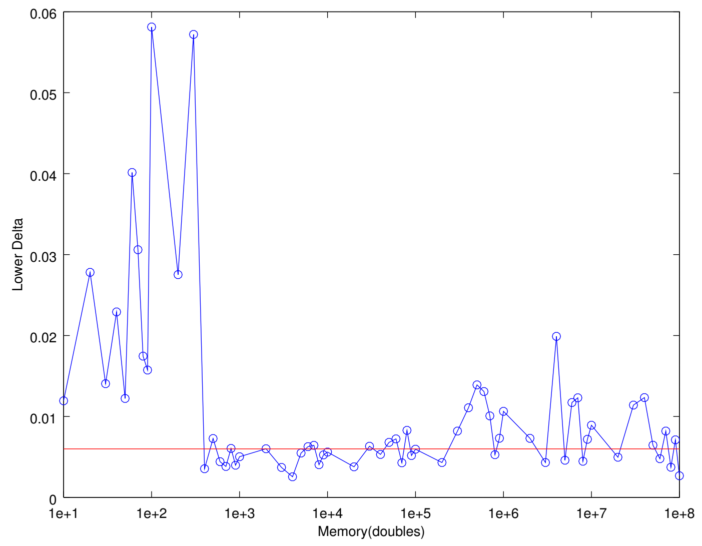


(a) Alphas from Lower Supply Functions

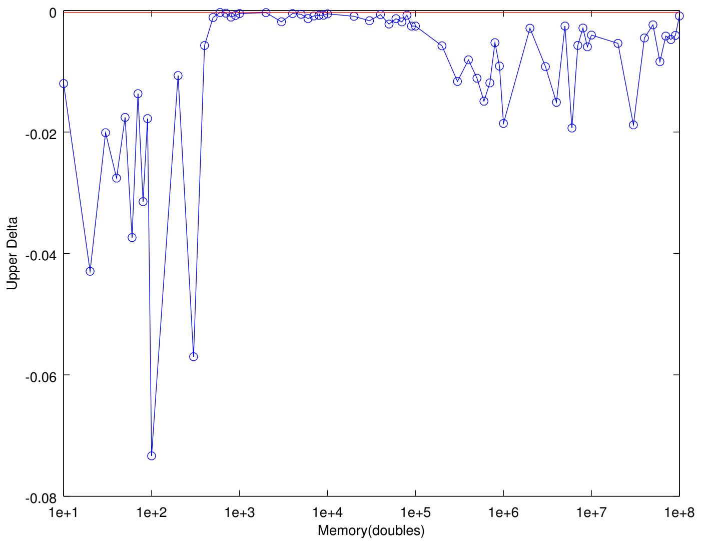


(b) Alphas from Upper Supply Functions

Figure 6.1: Plotted alphas from Supply Functions, Albert Tests.



(a) Deltas from Lower Supply Functions



(b) Deltas from Upper Supply Functions

Figure 6.2: Plotted deltas from Supply Functions, Albert Tests.

Figure 6.1 show the values for the upper and lower bound α values. The α values are around 0.998 while the memory allocation is below 10.000 doubles, then dips to around 0.965.

In Figure 6.2 the values for the upper and lower bound Δs can be seen. They reveal that the amount of memory used during the tests have an effect on the results. For memory allocation below 100.000 doubles the Δ values are low and close to constant and for higher memory values the Δ values start to show some irregularities.

In all of the above results an unexpected dip in α values can be seen for memory values below 500 doubles. This processor loss appears in all tests for these low amounts of memory, there is however yet no explanation for this behaviour. A guess would be that the usage of *malloc* for low memory values result in some unexpected behaviour due to it's optimistic memory allocation strategy[*malloc(3)* 2017]. Another guess would be that the minimum overhead for a malloc memory chunk is noticeable for small amounts of memory[Lea and Gloger, 2012]. The fact that this behaviour appears for as much as 500 doubles makes these guesses a bit far fetched.

The α values also show processor loss for higher memory values. There seem to be a memory usage where the thread is running optimally between 500 doubles and 100.000 doubles.This effect occurs during most tests with a varying amount of memory and can also be seen in figures 6.4 and 6.5.

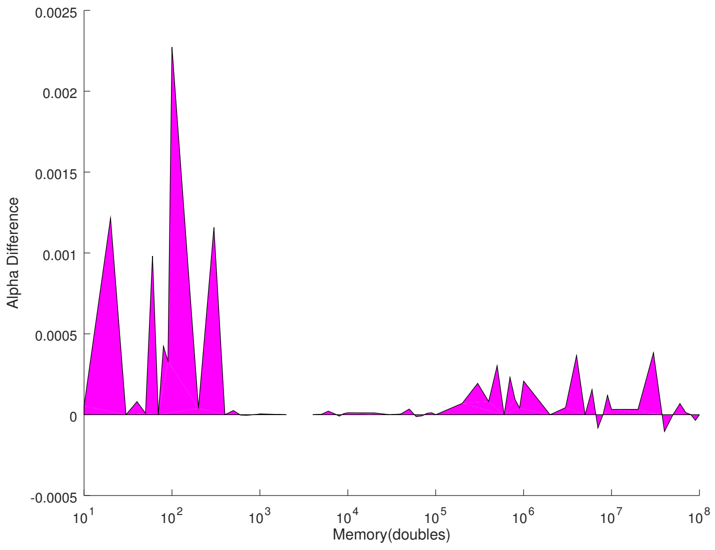
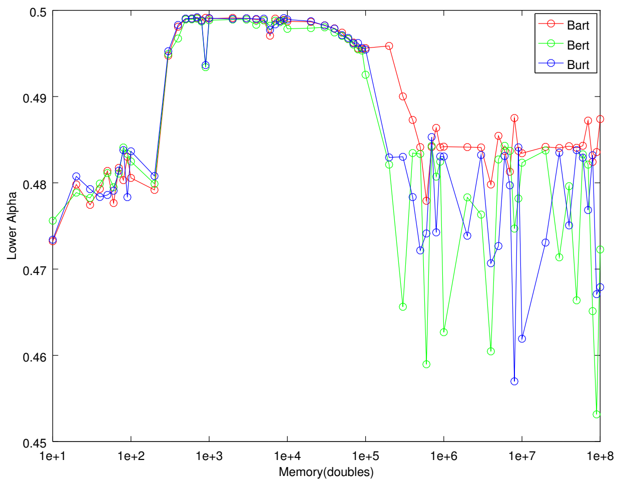


Figure 6.3: Difference between lower and upper alphas from the Albert tests.

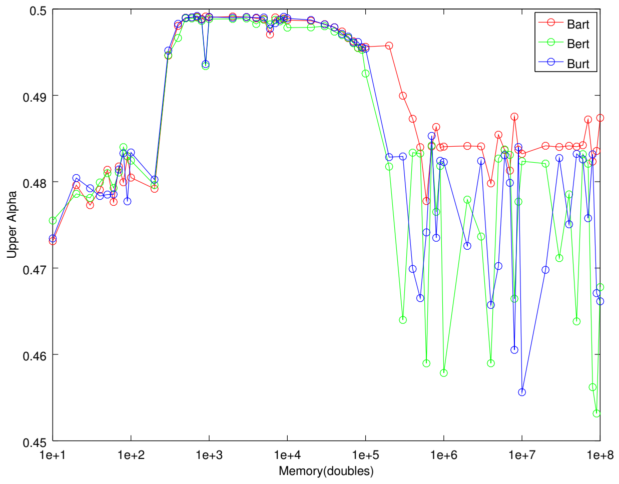
6.2 Bart, Bert and Burt Tests

The Bart, Bert and Burt tests were made to see how other threads running at the same time would affect a running thread with memory allocation. The additional threads had different configurations in each test where the memory allocations would differ.

The Bart tests were not expected to differ too much from the Albert tests since the additional thread did not have any memory allocation, however lower alphas were expected for both Bert and Burt where the memory allocation was added.

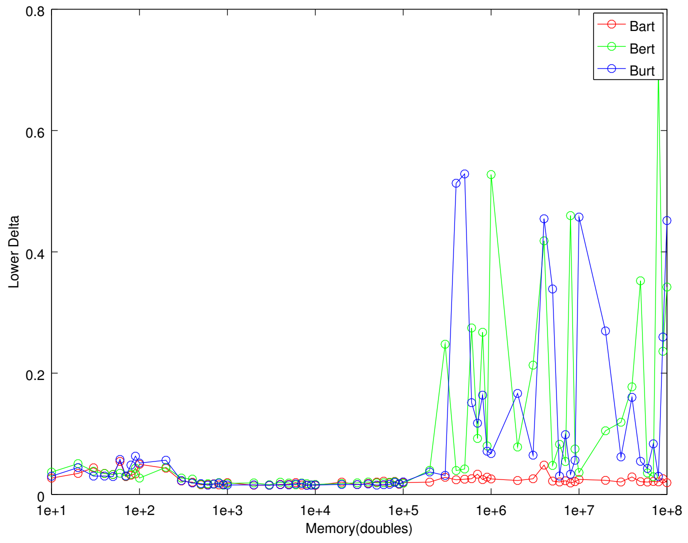


(a) Alphas from Lower Supply Functions

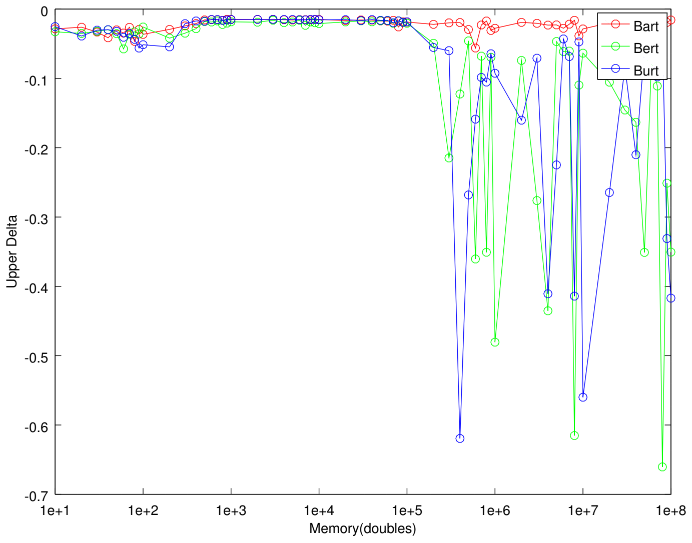


(b) Alphas from Upper Supply Functions

Figure 6.4: Plotted alphas from Supply Functions, Bart, Bert and Burt Tests.



(a) Deltas from Lower Supply Functions



(b) Deltas from Upper Supply Functions

Figure 6.5: Plotted deltas from Supply Functions, Bart, Bert and Burt Tests.

The behaviour in the Figures 6.4 and 6.5 are similar to the behaviour in Figures 6.1 and 6.2 from the Albert tests. There is a processor loss for memory allocation lower than 500 doubles and higher than 10.000 doubles and the α values have a high variation above 10.000 doubles.

The α variation for high memory is greater for the Bert and Burt test since the additional threads for these tests use memory allocation.

The change in the values always occur around a set amount of memory and can be associated with the cache size of the machine running the tests. This set amount of memory is close to the total cache size thus occurring when data needs to be stored in memory space accessed slower than the cache, i.e. the RAM memory.

The behaviour of these tests additionally confirms that our implementation is done in a correct matter. For an increased memory allocation for an application the α values should decrease, meaning that the memory causes delays when the threads are ready to run. The upper δ values should increase and the lower should decrease, since more delays are now possible.

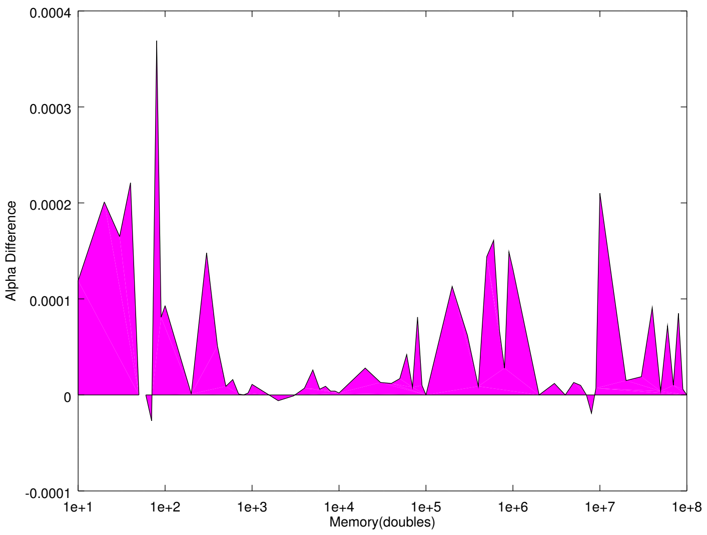


Figure 6.6: Difference between lower and upper alphas from the Bart tests.

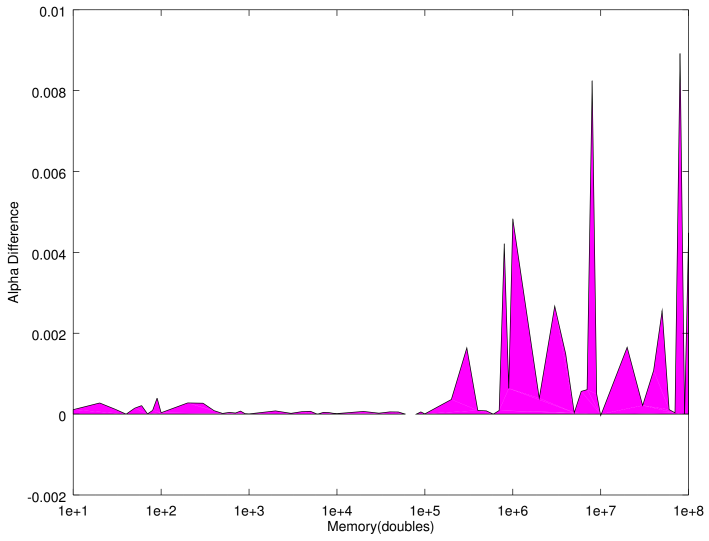


Figure 6.7: Difference between lower and upper alphas from the Bert tests.

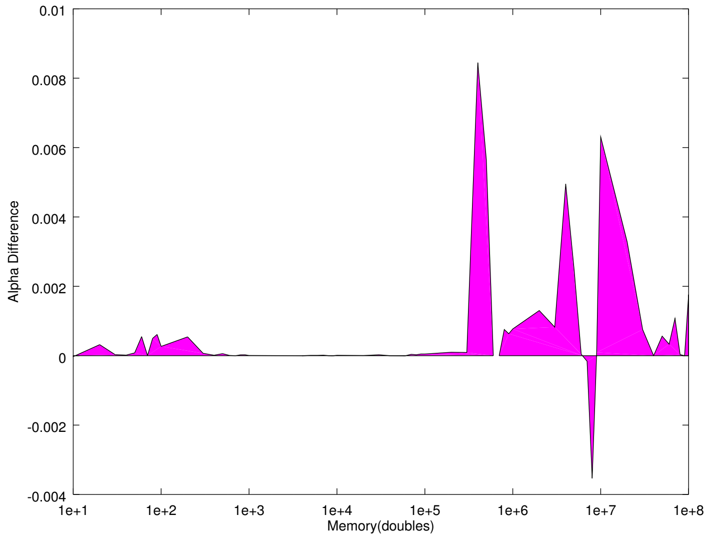
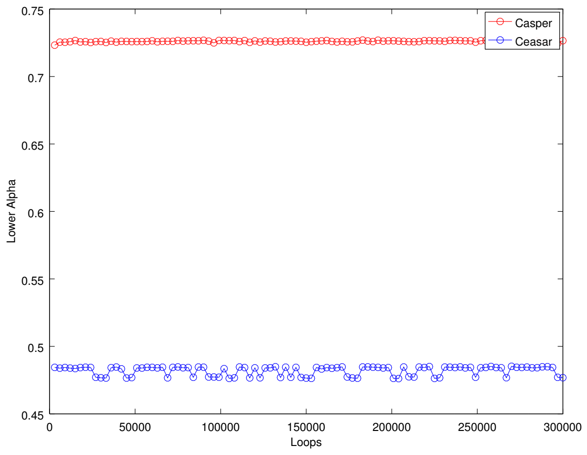


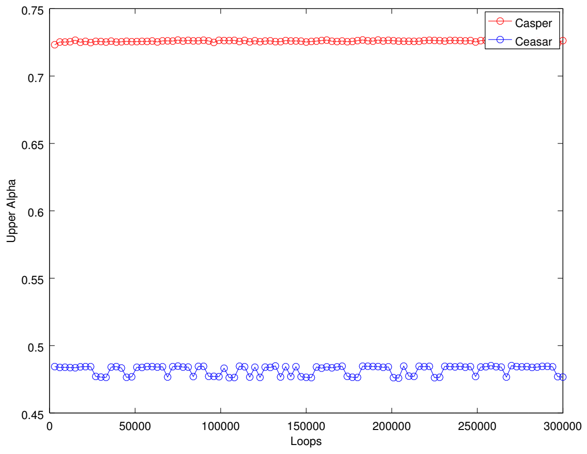
Figure 6.8: Difference between lower and upper alphas from the Burt tests.

6.3 Casper and Ceasar Tests

The Casper and Ceasar tests were made to evaluate the effect on a thread with memory allocation while other threads were being initialized and terminated. The fact that the original thread was using memory allocation was not expected to have an effect on the results where only the execution loops were modified.

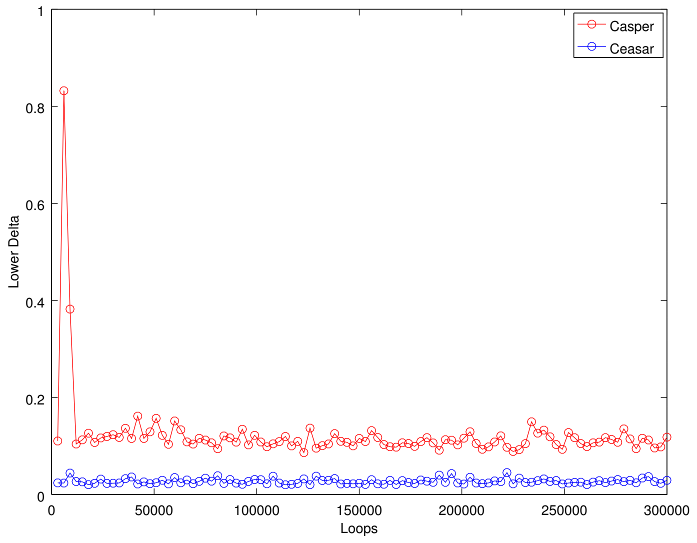


(a) Alphas from Lower Supply Functions

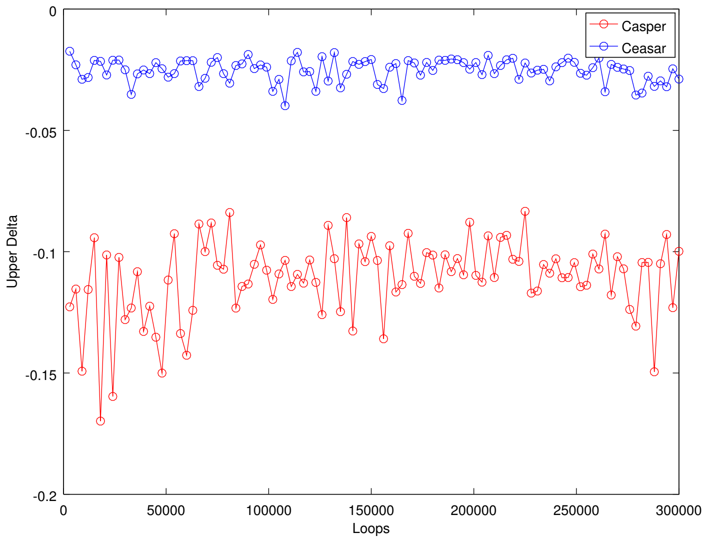


(b) Alphas from Upper Supply Functions

Figure 6.9: Plotted alphas from Supply Functions, Casper and Ceasar Tests.



(a) Deltas from Lower Supply Functions



(b) Deltas from Upper Supply Functions

Figure 6.10: Plotted deltas from Supply Functions, Casper and Ceasar Tests.

As seen in the Figures 6.9, 6.10, 6.11 and 6.12, both the α values are static at around 0.73 and the Δ values have static low values telling us that the changing loop values does not affect the amount of CPU used by the threads.

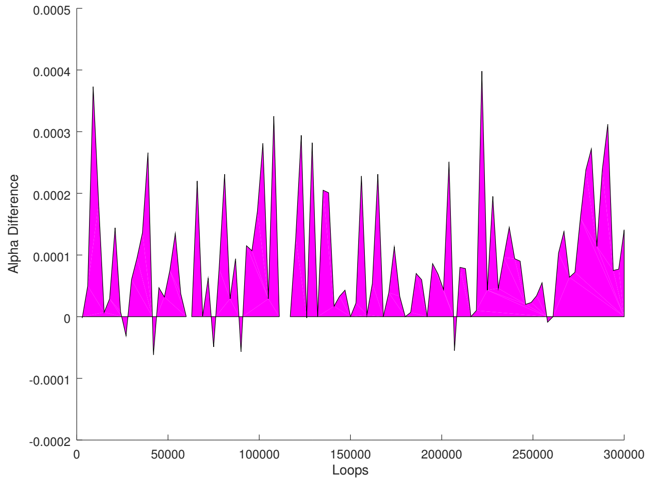


Figure 6.11: Difference between lower and upper alphas from the Casper tests.

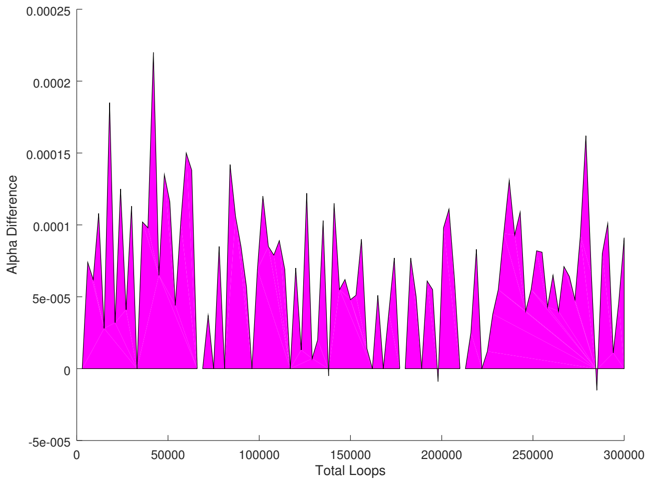
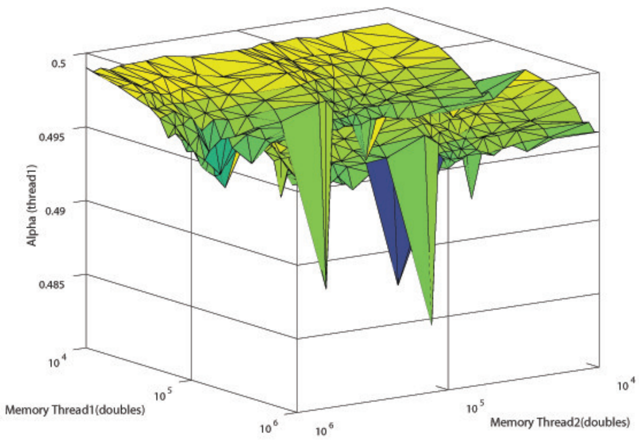


Figure 6.12: Difference between lower and upper alphas from the Ceasar tests.

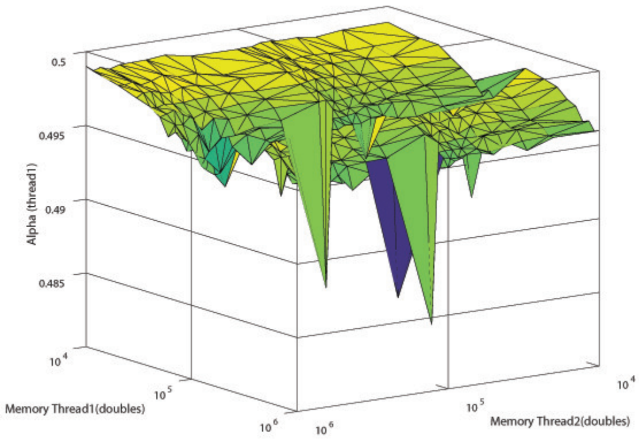
6.4 Dumble Tests

By running a large amount of scenarios for two threads with memory allocation a good visualization of the overall behaviour could be acquired. These tests also show how the measured thread is affected by different memory allocations in a secondary thread while the memory is kept constant in the measured thread.

A similar behaviour that was seen in the Albert, Bart, Bert and Burt test is expected when the total amount of memory would reach the cache size. A lowered α, a lowered Δ for the upper bound function and a increased Δ for the lower bound function.

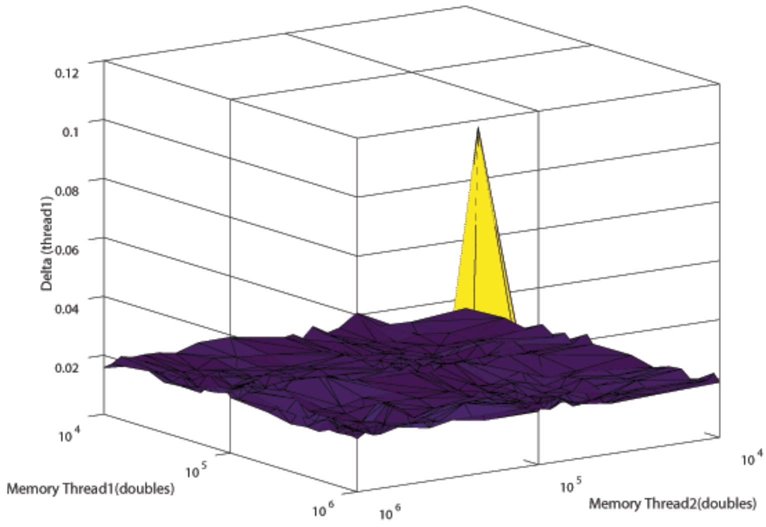


(a) Alphas from Lower Supply Functions

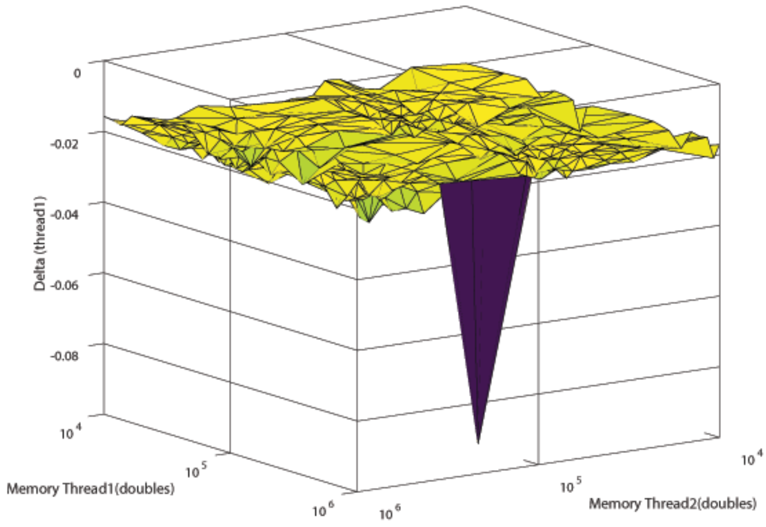


(b) Alphas from Upper Supply Functions

Figure 6.13: Plotted alphas from Supply Functions, Dumble Tests.



(a) Deltas from Lower Supply Functions



(b) Deltas from Upper Supply Functions

Figure 6.14: Plotted deltas from Supply Functions, Dumble Tests.

For low memory usage below 100.000 the α and Δ values are similar to the values in the Bart, Bert and Burt tests, being close to 0.50. The combined memory did however not seem to affect the thread in a substantial matter, only the individual memory allocation.

A dip in the α values were seen around 10.000 doubles, just like in the above tests, the values however did not stay low for higher memory values, but rose and stabilized between 0.49 and 0.50.

A similar outcome could be seen in the Δ plots where Δ was was being kept stable between at -0.02 for the upper and 0.02 for the lower Δ in all executions.

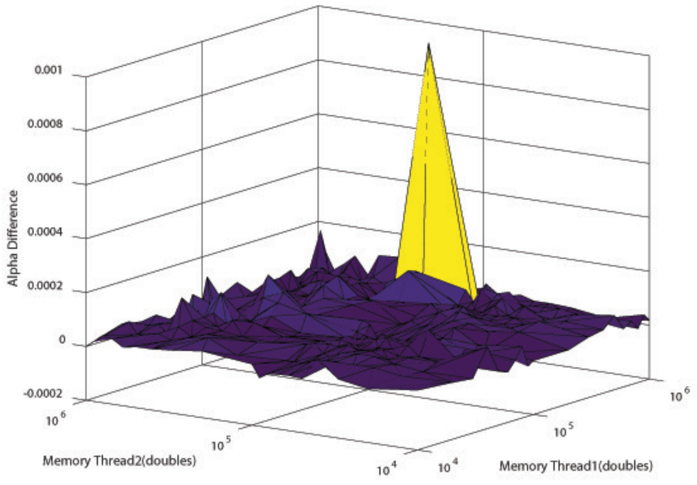


Figure 6.15: Difference between lower and upper alphas from the Dumble tests.

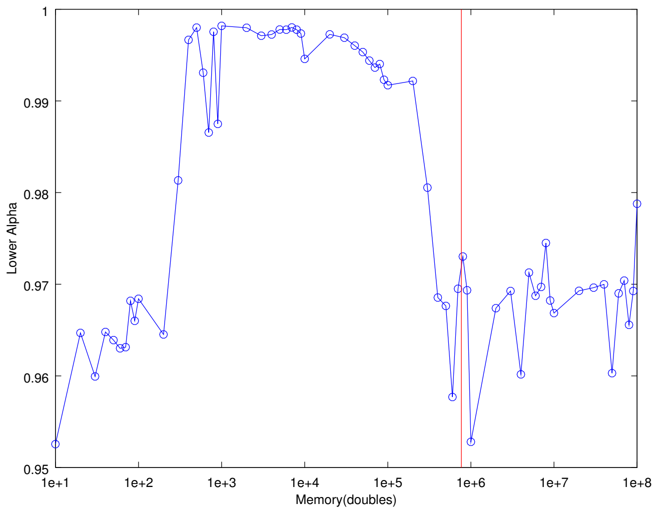
In Figures 6.3, 6.6, 6.7, 6.8, 6.11, 6.12 and 6.15 we can see the α-difference for the Albert, Bart, Bert, Burt, Casper, Ceasar and Dumble tests. In the α-difference plots belonging to the Albert, Bart, Bert and Burt tests the memory interference appears as well. The biggest difference can be found in the Bert and Burt tests and these only reach a difference of approximately 0,9%. When the difference rises above 5-10% the ability to provide an appropriate amount of resources to smoothly run the tasks without delays could be compromised.

The α differences that go below zero is only a result of test interference. These results are theoretically impossible when there is no limit on running the test over an infinite amount of time. This would indicate that the lower and upper supply functions intersected. Since the differences is very close to zero the interference does not have to be large to generate these results.

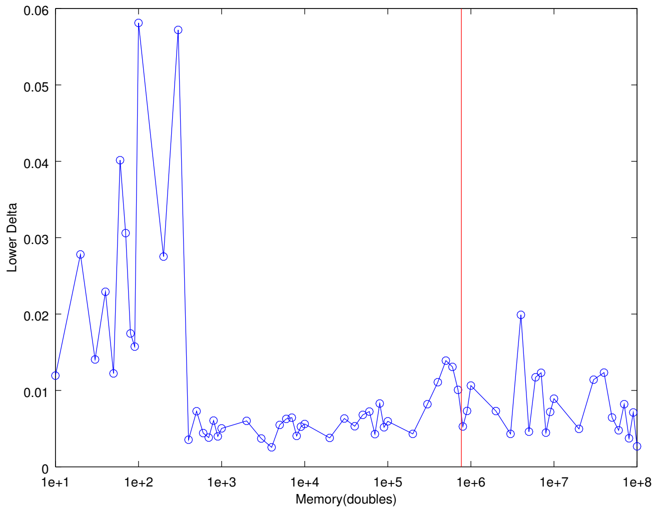
Additionally, in all figures belonging to the Dumble tests, Figures 6.13, 6.14 and 6.15, some distinct peaks can be seen. These could not be explained, but the exact values were gathered and can be found in Appendix B.

6.5 Cache Size Consideration

As stated in Section 3.1 the size of the cache is 6144 kB. Taking in consideration that the size of a double, that is used to allocate memory in the tests, is 8 bytes it will take 768,000 doubles to fill up the cache. The cache size is featured in the following plots.

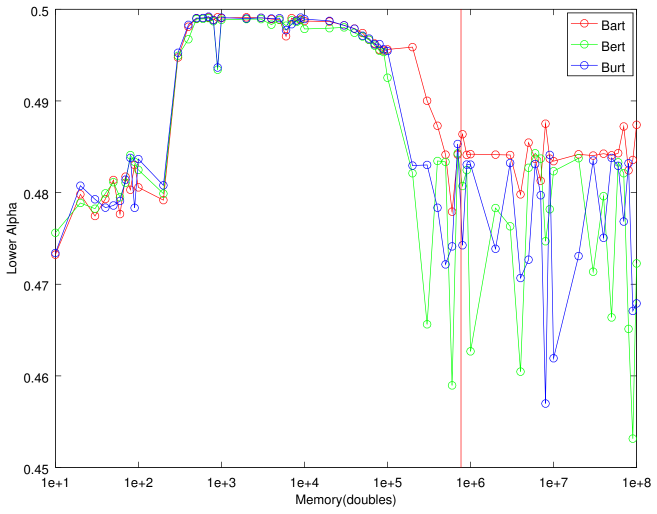


(a) Alphas from Lower Supply Functions

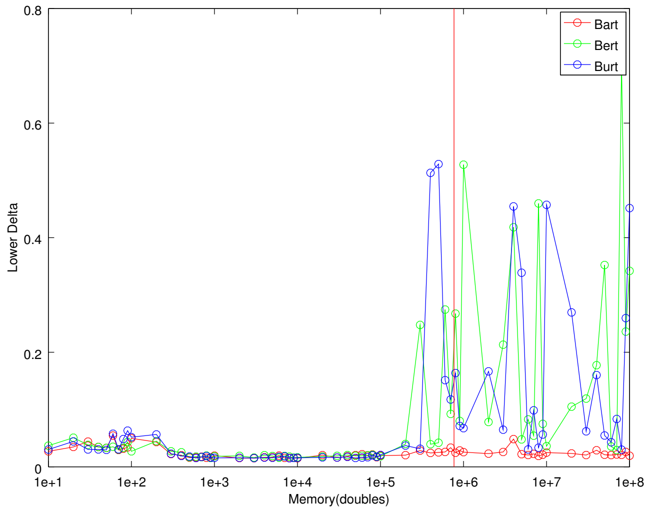


(b) Deltas from Lower Supply Functions

Figure 6.16: Plotted lower bound values with the total cache size featured as a red line, Albert Tests.



(a) Alphas from Lower Supply Functions



(b) Deltas from Lower Supply Functions

Figure 6.17: Plotted lower bound values with the total cache size featured as a red line, Bart, Bert and Burt Tests.

In Figures 6.16 and 6.17 the total cache size has been featured as a vertical red line. In these figures the change always occurs before the memory used reaches the cache size limit. Since other processes on the machine also uses the cache memory, the variance will not occur exactly at the cache size. By running the tests on a separate machine the interference from other processes is brought to a minimum. This means that the offset that can be seen in these tests are primarily caused by the operative system being run.

7

Conclusion

The purpose and goal of this thesis was to enhance the features of rt-bench by including memory usage on the threads and increase the area of application. The aim was to provide a more realistic execution of a simulated application, increasing the flexibility for testing the behaviour of platforms and set of threads.

As seen in the tests, the memory usage of the tests impact the behaviour at certain memory levels. These memory levels were anticipated and confirm the assumptions of realism in the simulations. When reaching storage limits, changes in behaviour are to be expected.

The change in behaviour can also be seen in the α-difference plots and it is also concluded that the α-difference is a highly relevant characteristic to consider when observing simulations on an execution platform. When the bound supply function curves keep diverging the estimation of an actual supply function will be impossible.

The importance of accurate models is high when making assumptions from the results of simulated runs in rt-bench and the possibility of simulating memory usage has made this a lot easier.

Bibliography

Blanquart, J.-P., J.-M. Astruc, P. Baufreton, J.-L. Boulanger, H. Delseny, J. Gassino, G. Ladier, E. Ledinot, M. Leeman, J. Machrouh, P. QuÃľrÃľ, and B. Ricque (2012). *Criticality categories across safety standards in different domains*.

Bovet, D. P. and M. Cesati (2005). *Understanding the Linux Kernel*. Covering version 2.6, 3rd. OâĂŹReilly Media, Inc.

Burns, A. and R. I. Davis (2016). *Mixed Criticality Systems - A Review*. Tech. rep. University of York, York, UK.

Buttazzo, G. C. (2011). *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer New York Dordrecht Heidelberg London.

Calandrino, J. M., H. Leontyev, A. Block, U. C. Devi, and J. H. Anderson (2006). *LITMUS^{RT}: A Testbed for Empirically Comparing Real-Time Multiprocessor Schedulers*. Tech. rep. The University of North Carolina at Chapel Hill.

Goel, N. and R. Garg (2012). "A comparative study of cpu scheduling algorithms". *International Journal of Graphics & Image Processing* **2**:4.

Katcher, D. I., H. Arakawa, and J. K. Strosnider (1993). "Engineering and analysis of fixed priority schedulers". *IEEE Transactions on Software Engineering* **19**:9.

Kerrisk, M., P. Zijlstra, and J. Lelli (2016a). *sched - overview of scheduling APIs*. [Accessed April 22, 2017]. URL: `http://man7.org/linux/man-pages/man7/sched.7.html`.

Kerrisk, M., P. Zijlstra, and J. Lelli (2016b). *sched - overview of scheduling APIs*. [Accessed April 23, 2017]. URL: `http://man7.org/linux/man-pages/man2/sched_setaffinity.2.html`.

Lea, D. and W. Gloger (2012). *Malloc implementation for multiple threads without lock contention*. source code. Version ptmalloc2-20011215. URL: `https://github.com/lattera/glibc/blob/master/malloc/malloc.c#L104`.

Lelli, J., G. Lipari, D. Faggioli, and T. Cucinotta (2011). *An efficient and scalable implementation of global EDF in Linux*. PhD thesis. Sant'Anna, School of Advanced Studies, Pisa, Italy.

Maggio, M., E. Bini, and J. Lelli (2016). “A tool for measuring supply functions of execution platforms”. RTCSA, pp. 39–48. DOI: `10.1109/RTCSA.2016.14`.

malloc(3) (2017). Linux man-pages. URL: `http://man7.org/linux/man-pages/man3/malloc.3.html`.

Mok, A., X. Feng, and D. Chen (2001). “Resource partition for real-time systems”, pp. 75–84. DOI: `10.1109/RTTAS.2001.929867`.

Nahas, M. (2008). *Bridging the gap between scheduling algorithms and scheduler implementations in time-triggered embedded systems*. PhD thesis. University of Leicester.

Shaw, Z. A. (2015). *Learn C the Hard Way: A Clear & Direct Introduction to Modern C Programming*. Addison Wesley, 2015.

The JSON Data Interchange Format (2013). [Accessed October 20, 2015]. Ecma International. Ecma International. URL: `http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf`.

TRACE-CMD(1), man page (2015). [Accessed November 10, 2015]. Linux man-pages. URL: `http://man7.org/linux/man-pages/man1/trace-cmd.1.html`.

Appendices

A

List of Support Files used with rt-bench

xlaunch.sh	- Run multiple tests automatically.
make_outputs.sh	- Collect result data from the first thread.
make_mult_outputs.sh	- Collect result data from multiple specified amount of threads.
gen_plot.sh	- Generate customized plot.
gen_3Dplot.sh	- Generate customized plot from input, specifically made for 3D-plots.
plot_template.m	- Template used by *gen_plot.sh* and *gen_3Dplots.sh*.

B

Deviations in Dumble tests

Memory (Thread 1)	Memory (Thread 2)	Alpha	Delta
70	100	0.484306	0.018046
700	100	0.484065	0.020903
900	200	0.487457	0.116806

Table B.1: Supply Lower Bound Functions, Diverging Alphas

Memory (Thread 1)	Memory (Thread 2)	Alpha	Delta
70	100	0.484304	-0.019425
700	100	0.484006	-0.019347
900	200	0.486524	-0.092376

Table B.2: Supply Upper Bound Functions, Diverging Alphas

Memory (Thread 1)	Memory (Thread 2)	Alpha	Delta
900	200	0.487457	0.116806

Table B.3: Supply Lower Bound Functions, Diverging Deltas

Memory (Thread 1)	Memory (Thread 2)	Alpha	Delta
900	200	0.486524	-0.092376

Table B.4: Supply Upper Bound Functions, Diverging Deltas

Lund University **Department of Automatic Control** **Box 118** **SE-221 00 Lund Sweden**	*Document name* MASTER'S THESIS *Date of issue* October 2017 *Document Number* TFRT-6043
Author(s) Zsolt Demeter	*Supervisor* Martina Maggio, Dept. of Automatic Control, Lund University, Sweden Karl-Erik Årzén, Dept. of Automatic Control, Lund University, Sweden (examiner)

Title and subtitle

RT-Bench, Improved Understanding of Application Performance with Memory Storage

Abstract

By implementing efficient and smart schedulers in our software systems with multiple threads we can make applications run faster and much more efficiently. There is however a lot of caution when adopting and implementing scheduling algorithms, like limited preemptive scheduling or PD2, due to the uncertainty they may cause on advanced and complex systems. In fact, most algorithms are tested to produce advantages in specific situations. This is one of the reasons why there is a gap between the theoretical scheduling development and the actual schedulers implemented in real operating systems. One way to close the gap is to derive precise guarantees for the implementation of scheduling algorithms, which is the purpose of rt-bench.
Rt-bench calculates characteristic values for a specified scheduling algorithm and a specific task set, mainly in form of supply bound functions based on the execution of the task set on a Linux-based hardware platform. The characteristics can vary depending on the system and the setup, therefore these are used to compare complete execution platforms rather than single algorithms.
This thesis focuses on extending rt-bench to increase the realistic behaviour of the simulation of the application behaviour. Before this thesis, rt-bench could simulate computations but not memory handling. Simulating memory management is necessary to create realistic models and the purpose of this thesis is to introduce this memory usage in rt-bench.
The results show a clear performance drop before the model reaches a memory level equal to the cache size, due to other processes also using the cache memory. This behaviour is what was expected and confirms that the implementation is sufficient for measuring and evaluating performance offered by different platforms.

Keywords

Classification system and/or index terms (if any)

Supplementary bibliographical information

ISSN and key title 0280-5316		*ISBN*
Language English	*Number of pages* 1-71	*Recipient's notes*
Security classification		

http://www.control.lth.se/publications/