periodica polytechnica

Electrical Engineering
52/3-4 (2008) 145–152
doi: 10.3311/pp.ee.2008-3-4.03
web: http://www.pp.bme.hu/ee

RESEARCH ARTICLE

Test component assignment and scheduling in a load testing environment

Levente Erős / *Ferenc* Bozóki

Received 2008-07-22

Abstract

In this paper we introduce two major problems from the field of load (or performance) testing and our solutions for them. When testing the performance of a device (System Under Test – SUT), the test environment executes many software entities (the so-called test components) on the hosts of the test environment (testing hosts). Our goal is to maximize the load on the testing hosts by assigning the test components to them closely to optimal. The first problem to be solved is, thus, a special case of the task assignment problem for which many algorithms have been developed. Our solutions presented in this paper are, however, optimized for distributing load testing traffic in the case of which the possibilities and restrictions to be taken into account are very different from those of the classical task assignment case. The other problem we deal with is how to schedule test components running on the same testing host. Most of the papers written on scheduling focus on the characteristics of the generated load, but not on the way of generating it. These papers usually assume that the load can be generated by improving hardware resources. In this paper, however, we introduce a model and an algorithm which improves the efficiency of scheduling in a load testing environment with way less hardware resources. The algorithm is based on our novel concept of virtual threads. Our simulations have shown that by applying our solutions, the efficiency of load testing can be significantly increased.

Keywords

load testing · task assignment · scheduling

Acknowledgement

This work was supported by our supervisor, Tibor Csöndes PhD of Ericsson Hungary Ltd.

Levente Erős
BME, Hungary
e-mail: eros@tmit.bme.hu

Ferenc Bozóki
Test Competence Center, Ericsson Hungary Ltd., Hungary
e-mail: ferenc.bozoki@ericsson.com

1 Introduction

Nowadays testing is becoming a more and more important phase of the development process, since the earlier a fault is found in a product, the easier and less expensive it is to correct it. Many different kinds of tests can be applied to a device. A conformance test checks whether the System Under Test (SUT) corresponds to its functional specifications, that is, whether it implements the theoretical state machine it should according to its specifications [1]. During a conformance test the test environment simulates a single user connected to the SUT. Once the SUT is found to be corresponding to its functional requirements, it has to be tested by another important aspect, that is, whether it can deal with all the load it has to handle once it is placed and starts to operate in its latter real-life environment. This kind of testing is called load (or performance) testing [2]. A load test simulates the real-life environment of the SUT containing many users, pushes the generated load up to the maximal level the SUT has to be able to handle and examines the behavior of the SUT in this extreme situation. Fig. 1 illustrates the differences between the number of users in a conformance testing and in a load testing environment.

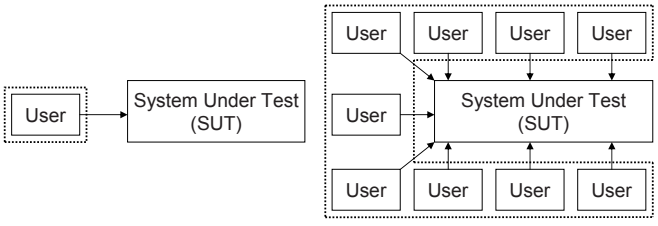

Fig. 1. Conformance and load testing environments

Unfortunately, the number of hosts in the test environment (testing hosts) is way less than the number of hosts the SUT has to serve in the real-life environment. Thus, the real-life hosts are emulated by software entities, the so-called test components, and each testing host executes more than one test component. At this point two important questions emerge, namely, how should the test components be assigned to the testing hosts to be able to simulate the real-life environment faithfully by as few test-

ing hosts as possible, and inside a testing host, how should the execution of test components be scheduled to minimize the overhead caused by context switching. In this paper we discuss and give solutions for these two problems.

The rest of the paper goes as follows: In Section 2 we discuss the problem of assigning test components to testing hosts. In Section 2.1 we give a formal definition for the problem. In Section 2.2 we introduce the main idea our test component assignment algorithms are based on. In Section 2.3 we introduce our test component assignment algorithms, and in Section 2.4 we close the discussion of test component assignment with some simulation results. In Section 3 we discuss our sceduling algorithm. In Section 3.1 we introduce a new archiecture for generating the desired load. In Section 3.2 we present the possible bottleneck of the architecture and in Section 3.3 we are going to describe a practical example to demonstrate the problem caused by it. Finally, in Section 3.4 we introduce our scheduling algorithm that solves the problem, and we summarize our results in scheduling in Section 3.5. We conclude our paper with a few words on our future work in Section 4.

2 The test component assignment problem

As mentioned earlier, when assigning test components to testing hosts, our goal is to minimize the number of necessary testing hosts by assigning the test components to testing hosts closely to optimal. The task of minimizing the number of testing hosts equals to the task of maximizing the load taken by them. Carrying out an optimal assignment would, however, be NP-hard, thus a heuristic algorithm is needed. Fig. 2 shows a typical load testing environment with many parallel testing hosts connected to the SUT and with the so-called Main Controller supervising the testing hosts and assigning the test components to them.

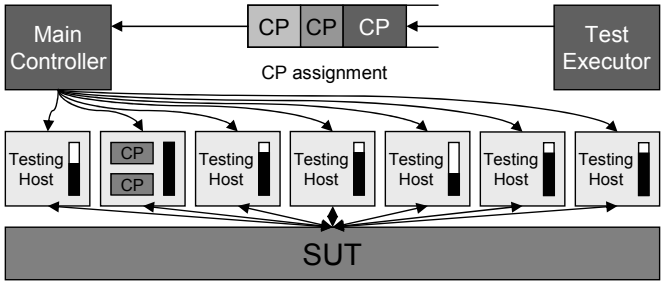

Fig. 2. Details of a load testing environment

The problem described above might remind us to task assignment problems of distributed systems in the field of which many papers have been written [3], [4], [5], but our case is way more specific, since our algorithm has to take the specialities of load testing environments into consideration.

In this section we present a heuristic algorithm for the solution of the above mentioned problem. The presented algorithm works on a formal model, but the goal of its final version will be to process test source codes written in TTCN-3 (Testing and Test Control Notation - 3) language [11].

2.1 Formal problem definition

The task assignment problem can be defined as follows:

$TH = \{TH_i | i = 1, ..., m\}$ is the set of testing hosts. Each one of the testing hosts is described as follows: $TH_i = (TCH_i, \{FCH_i^t | t = 1, ..., t_{max}\})$. TCH_i is the *total capacity* and FCH_i^t is the *free capacity* of TH_i in time slot t. (A time slot is the smallest unit of time in our model.) Furthermore, there is an $m+1^{st}$ testing host for modeling test component dropping: $TH_{m+1} = (\infty, \{\infty, ..., \infty\})$. This means that test components that get dropped are assigned to an imaginary testing host with infinite capacity according to the model.

$CP = \{CP_i | i = 1, ..., n\}$ is the set of test components. A test component is described as follows: $CP_i = (FSC_i, LCC_i, TIDC_i, RCC_i)$, where FSC_i is the *starting time*, LSC_i is the *ending time* of test component i, $TIDC_i$ identifies the *type* of the test component, while RCC_i is its *required capacity*. The *type* of the component is the type declared in the TTCN-3 source code of the load test. Test component instances of the same type have similar charasteristics, for example, very similar (in our simple case, identical) capacity requirements, and similar running times.

Our objective is to make single-valued $CP \rightarrow TH$ (test component to testing host) assignments, so that the largest possible load is generated by the test environment, formally, $\sum_{t=1}^{t_{max}} \sum_{i=1}^{m} (TCH_i - FCH_i^t)$ is maximal. When making the assignments, there are two constraints. First, testing hosts cannot be overloaded, formally $FCH_i^t \geq 0$, where $i = 1, ..., m$ and $t = 1, ..., t_{max}$. The second constraint is the following: each test component can only be assigned to one testing host, so it is not possible to reassign a component during its execution. The reason for this is that if a test component is reassigned during its execution the authencity of the simulation of the real-life environment would be at stake, since a time gap would appear in the execution of the reassigned component.

2.2 Main idea

Since the goal is to stress the SUT by the maximal load it has to be able to handle, several test components are needed to be run in parallel. Consequently, the running time of test components is long compared to the time elapsed between their execution, since by many components starting and ending all the time, the load generated by the test environment could not be pushed up to its desired level. In a more abstract way, once the running components can produce the desired load, no other components are executed "for a while", thus components are rather running "together". Fig. 3 illustrates a load testing traffic pattern.

Based on this property, the time can be splitted up into time frames with length W and consider the problem as a series of bin packing problems [6], regarding testing hosts as the bins and

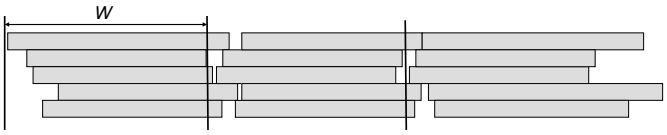

Fig. 3. Load testing environment

test components starting in the same time frame as the goods to be assigned to bins. However, the $CP \rightarrow TH$ assignments for time frame k can be first made at the end of the time frame when the parameters of all the components of time frame k are known. Thus, the actual execution of the test components has to be delayed at least until the end of the time frame, but to keep the original shape of the load testing traffic pattern, the starting times of each of the test components of time frame k (and the entire time axis) will be delayed by W (see Fig. 4).

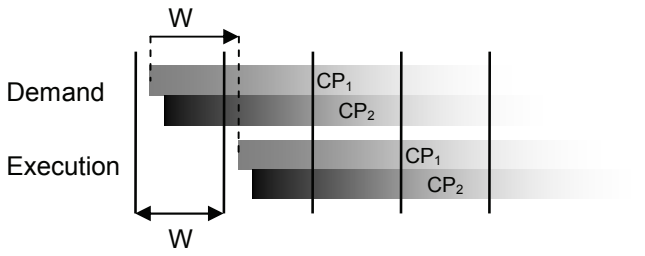

Fig. 4. Delayed execution of test components

There is one more thing left to be mentioned. Unlike the classical bin packing problem, the free capacity of testing hosts is not constant within a time frame, as formerly started test components can terminate and new components are started in the time frame. Thus, when applying a bin packing algorithm to a time frame, this property has to be taken into consideration.

The above described test component assignment method can only be applied, if three constraints are met. The first constraint is that delaying the execution of all the test components does not affect the outcome of the test. This constraint is true if the outcome of the execution of one component does not affect the execution of another component. The second constraint requires us to know the running times of test components, while the third constraint assumes that the capacity requirements of the different component types are known (with component instances of the same type having identical capacity requirements).

2.3 Two possible solutions for the task assignment problem

In this section we are going to introduce two solutions for the problem defined above, both based on the time framing approach.

2.3.1 The bin packing-based approach

The first solution regards test components starting in the same time frame as goods to be packed into bins (testing hosts). It extends already existing bin packing algorithms. We tested this solution using the First Fit Decreasing [6] algorithm for assigning test components to testing hosts within a time frame.

Let us now look at how the test component algorithm works in time frame k. The algorithm has three inputs, the required capacities of test components starting in time frame k, the capacities of the testing hosts for each time slot in time frame k, and the bin packing algorithm, the FFD in our case. First, the test components of frame k are ordered by their required capacities in descending order. Then the test component (let it be test component i) with the largest required capacity is assigned to the first testing host the free capacity of which is larger or equal than the required capacity of test component i, in each of the time slots of time frame k (when test component i is active). The test component assignment algorithm goes on with the test component that has the second largest required capacity, and so on to the test component with the smallest capacity.

It is important to note, that when seeking for a testing host with enough capacity the "next" testing host is the next testing host among testing hosts 1 to m. Host $m+1$ is only chosen if there are no other testing hosts the component fits into.

2.3.2 The binary programming-based approach

The second solution describes the test component assignment problem with a series of binary programming problems (one for each time frame). But before going on with introducing this solution the following variable and matrices have to be defined:

$AC_u^v = 1$, if $FSC_i \leq v \leq LSC_i$, 0 otherwise, that is AC_u^v indicates, whether CP_u is active in time slot v, or not.

$\mathbf{RCap}^{t_{max} \times n} = [r_{uv}]$, where $r_{uv} = AC_v^u$, that is, this matrix stores the required capacities of test components for each time slot (and stores 0s where the corresponding test components are inactive).

$\mathbf{Sel}^{n \times (m+1)} = [s_{uv}]$, where $s_{uv} = 1$, if test component u is assigned to testing host v, 0 otherwise.

$\mathbf{TCap}^{t_{max} \times (m+1)} = [f_{uv}]$, where $f_{uv} = TCH_j$, that is **TCap** stores the total capacities of testing hosts.

$\mathbf{FCap}^{t_{max} \times (m+1)} \mathbf{TCap} - \mathbf{RCap} \cdot \mathbf{Sel}$, that is **FCap** stores the momentary free capacities of testing hosts. (This matrix has to be recalculated each time the assignments of a time frame are made - see more on this later.)

The matrices defined above are all global, that is they are used for keeping some kind of a global state of the execution of the test component assignment algorithm, that is, the effects of the assignments that have already been made. This global state of the algorithm will become clear later.

The algorithm works as follows:

First, $s_{uv} := 0$ for all of the elements of **Sel**. Consequently, in the beginning, **FCap** = **TCap**. As we will see, **Sel** and **FCap** will be formed time frame by time frame along the execution of the algorithm.

As mentioned earlier, test components starting in the same time frame are handled and assigned together. This means that the algorithm consists of $\lceil \frac{t_{max}}{W} \rceil$ iterations.

In the k^{th} iteration, that is, when processing the k^{th} time frame, or more precisely, the test components that are executed in time frame k, three "local" matrices are defined in addition to the global ones defined earlier:

$\mathbf{RCap}^k = [r^k_{uv}]$ and $r^k_{uv} = r_{uv}$, where $(k-1)\cdot W + 1 \leq u < k \cdot W$ and $v : TH_v \in \{ZH_z | (k-1)\cdot W + 1 \leq FSC_z < k \cdot W\}$, that is $\mathbf{RCap}^k$ is a sub-matrix of **RCap** containing the time slots in time frame k and the test components that have to be started in time frame k. The dimensions of the matrix are, therefore, W rows (at maximum, since in the last time frame it is t_{max} mod W) and as many columns, as many test components have to be started in time frame k. Let the set of these test components be denoted by CP^k.

$\mathbf{Sel}^k = [s^k_{uv}]$ is a matrix that contains the test component to testing host assignments of time frame k. The dimensions of the matrix is $|CP^k| \times (m+1)$. As we will see shortly, the rows of **Sel** that correspond to the test components to be started in time frame k will be defined by the rows of $\mathbf{Sel}^k$.

$\mathbf{FCap}^k = [f^k_{uv}]$, where $f^k_{uv} = f_{uv}$ and $(k-1)\cdot W + 1 \leq u < k \cdot W$ and $v = 1, ..., m+1$, that is $\mathbf{FCap}^k$ contains those rows of **FCap** which correspond to the time slots of time frame k.

Let us denote the number of time slots in time frame k by W'. (As mentioned earlier, W' equals to W, except in the last time frame, where it can be less.)

Among the global matrices, thus, there are two input matrices, namely **RCap** and **FCap**. In each iteration we take the sub-matrices of these two, namely $\mathbf{RCap}^k$ and $\mathbf{FCap}^k$, as described above, and compute $\mathbf{Sel}^k$, which is the output matrix of time frame k by solving the binary programming problem with the following objective function:

$$\min_{\mathbf{Sel}^k} \sum_{u=1}^{W'} \sum_{v=1}^{m} (\mathbf{FCap}^k - \mathbf{RCap}^k \cdot \mathbf{Sel}^k)_{uv} \quad (1)$$

and the following constraints:

$$\mathbf{RCap}^k \cdot \mathbf{Sel}^k \leq \mathbf{FCap}^k \quad (2)$$

$$[1...1] \cdot \mathbf{Sel}^{k^T} = [1...1] \quad (3)$$

$$s^k_{uv} = \{0, 1\} \quad (4)$$

The objective function (Eq. 1) formalizes our desired goal, that is, minimizing the free capacity on testing hosts 1 to m (but not for testing host $m+1$, as it has an infinite capacity and is used for collecting dropped test components). Constraint (Eq. 2) forbids the overloading of testing hosts, constraint (Eq. 3) ensures that each testing host is assigned to exactly one testing host (to testing host $m+1$ if dropped). Constraint (Eq. 4) limits the values of the elements of $\mathbf{Sel}^k$ to 0 or 1.

Once the binary program is solved, that is $\mathbf{Sel}^k$ defining the assignments made in time frame k is calculated, the corresponding rows of the global matrix **Sel** are updated by the rows of $\mathbf{Sel}^k$. Then the global matrix **FCap** is recomputed, so it is going to contain the remaining free capacities of testing hosts (the capacities of testing hosts that the still active test components assigned in time frames 1 to k left free). This way, when getting $\mathbf{FCap}^{k+1}$ from **FCap** for the next time frame and assigning the test components of time frame $k+1$, the effects made by previously assigned, but still running test components on the free capacities of testing hosts will be taken into consideration.

Test components that were originally meant to be started in time frame k are going to be started in time frame $k+1$ with a delay of W, and similarily, the test components assigned in time frame $k-1$ were started in time frame k with a delay of W.

The algorithm goes on like this until the last time frame. One might wonder why time frames are used in this binary programming approach, since we could define the whole problem with the global matrices, solve the global problem and get a solution that is closer to optimal than the one we get by solving smaller problems in each of the time frames. This global solution has unfortunately two drawbacks. First, it would mean a delay by t_{max} which is not too much desirable. Second, the complexity of solving the global problem is huge compared to that of the smaller problems solved in each time frame, even if using an efficient method for solving the binary program. In the case of the binary programming approach, therefore, the most important factor of adjusting the time frame size (window size) is the complexity of the assignment.

2.4 Results

For examining our task assignment heuristics we implemented a simulator. In this section we introduce the simulation results of the bin packing-based algorithm.

We examined the algorithm by comparing the average load level on testing hosts with different window sizes to that of the naive test component assignment algorithm. The naive algorithm assigns each incoming test component to the first host with enough free capacity. Fig. 5 shows the simulation results, that is the average utilization of testing hosts (vertical axis) for different window sizes (horizontal axis).

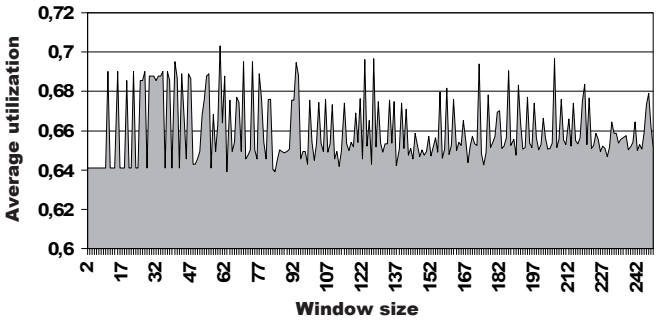

Fig. 5. Simulation results of the bin packing-based task assignment algorithm

The naive solution for test case assignment corresponds to the window size of 1 at which the average utilization is 64%.

By selecting an appropriate window size, the average utilization can be significantly raised compared to this naive solution.

3 Scheduling in a load testing environment

Most of the papers that are written in the field of scheduling mostly focus on scheduling in an environment where the cost of context switching is not negligible [9], [10]. Many operating system schedulers use binary balanced trees to optimize this task but still, all of these algorithms assume that before and after switching, different contexts have to be loaded in order to have the very same environement as it was during the previous execution period. Some of these schedulers are preemptive, that is, they can interrupt the execution of a particlar thread to switch to another one. There are, however, many threads that can not be interrupted, so a non-preemptive algorithm might be needed. In our case the cost of the context switching is zero since the context is stored in a common data structure, and the threads can not be preempted. In the following we introduce an algorithm that works with these two restrictions.

3.1 The architecture

The SUT is usually an element of a telecommunication network. It can be a network node in a GSM network [7]. In our example it is a Mobile Switching Center (MSC). The challenge in this case is to create an optimal architecture which is able to emulate numerous users with a minimal resource usage. During the conformance testing phase most of the users are realized by POSIX threads, and the operating system's scheduler (OS scheduler) schedules them. The one user – one thread approach works fine for a couple of users, but when it comes to load testing, where there are usually tens of thousands of users, most of the OS schedulers will spend as much time with context switching between the different users' processes as with the execution of the useful code. Thus, a smarter solution is needed, where a single thread is able to emulate numerous users. This way the emulation of, for example, 10000 users could be done with only 10 threads, each one handling around 1000 users. The multiple users – one thread architecture is based on the Finite State Machine theory [8]. Each emulated user is realised by an FSM (Finite State Machine). Each FSM can be described by a quadruple (see Eq. 5).

$$FSM = (s0, S, E, F) \tag{5}$$

Where

- $s0$ is the initial state
- S is the set of states
- E is the set of events
- F is the set of transition functions.

When an event $e_z \in E$ occurs the FSM changes its state from $s_x \in S$ to $s_y \in S$ and executes a transition function $f_h \in F$. In our case the transition functions are sequences of actions (send a message, etc.). When an event occurs, the sequence of actions assigned to it is executed. This sequence cannot be interrupted by other actions. To avoid the blocking of the thread, and thus, all of the FSMs, it is not allowed to use any blocking statements here (e.g. blocking while waiting for a message). Every thread has a database storing all the relevant information about the users on the thread (each record of this database belongs to a user, or more specifically, the context of a user). This way context switching is reduced to changing an index. Each thread contains an event queue as well (see Fig. 6). It stores all the incoming messages in the order of reception. Each message contains some information that is necessary to associate the message with the destination user. Processing such events means dispatcing them to the users they are destined to, while these destination users execute the corresponding actions.

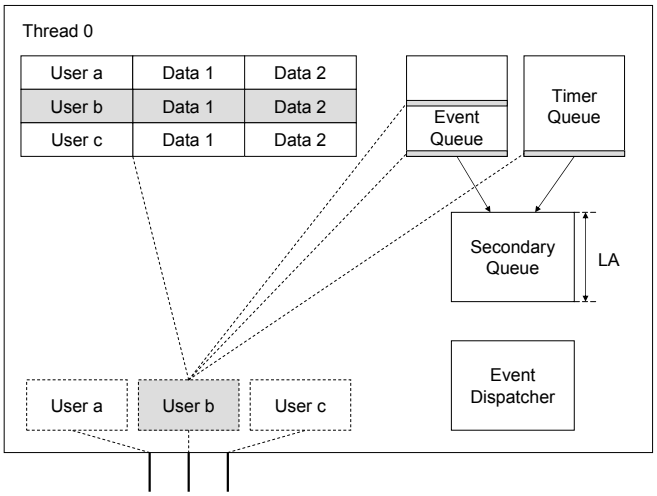

Fig. 6. Architecture of the test system

3.2 The bottleneck of this approach

In our environment the cost of context switching is close to zero, but the users still need to be scheduled somehow. Many OS schedulers handle regular threads, but in their case the cost of context switching is not even close to zero [9], [10]. Most of the telecommunication standards contain timing restrictions and in this environment all of the timing restrictions are handled by the timer queue. When a thread needs to use a timer, it places a timer event in this queue and when a timer expires in the queue, it dispatches an event to the corresponding user. Therefore we must distinguish between two different types of timers. One serves as a wait function, instructing the user to wait some time before moving on with the execution. The other serves as a guard timer. If there are no incoming messages within a specified time, an error handling method should be invoked. Usually the timer queue contains multiple timer events ordered by the remaining values of the timers, thus the one to expire first is always the first in the queue. When a timer expires it dispatches an event, which is interpreted by the FSM to whom it belongs. Sometimes the action associated with a timer event takes too

much time to serve and the rest of the process is delayed. We also would like to avoid the situation when the SUT had sent the response in time, but because of the internal structure of the test system, it is processed too late resulting in a false outcome (or verdict) of the test.

3.3 A practical example

In this section we present an example related to the GSM world. The example is an extract of some longer transactions. Let us suppose that we are testing the Base Station System Application Part (BSSAP) protocol of a Mobile Switching Center (MSC) [7]. We emulate 10 Mobile Stations (MS), 5 of which initiate a Location Update (LU) and the other 5 of which terminate a mobile call (MTC). Let us suppose that during the transactions we have reached a point when the LU users send a ChiperMode_Complete message, and wait for the TMSI_Reallocation_Cmd to arrive within 100 milliseconds. To respond they send a TMSI_Reallocation_Complete message in 50 milliseconds, etc. The other five users send out a Paging_Response message and wait for an authorization request to arrive. If the authorization request (Auth_Req) arrives within one second they count and send the authorization response (SRES). The latter definitely takes more resources and time. In case of success the SUT will send Chiper_Mode_Cmd message, within one second, etc. We have 10 FSM instances to realize the users, five for the first group and five for the second group. The first group of FSMs has the following two events:

- received_TMSI_reallocation_cmd,
- timer_expired.

They have two sequences of actions:

- send_ChiperMode_Complete, place_TMSI_timer_to_the_timer_queue,
- send_TMSI_reallocation_complete.

The other types of FSMs also have two events:

- received_auth_request,
- timer_expired,

and two sequences of actions:

- send_paging_response, place_authtimer_to_the_timer_queue,
- count_ SRES, send_auth_response.

Supposing that we have reached the described session of the communication, five FSMs send a ChiperMode_Complete message, five send a paging response, and all of them place their timer events in the timer queue. The timer queue will contain five entries for 100 milliseconds, and five for one second. All the responses of the SUT will be delivered to the event queue. To handle a TMSI_reallocation_cmd event, a TMSI_Reallocation_Complete message should be dispatched and the FSM should send out a TMSI_Reallocation_Complete message. On the other hand, to handle an Auth_Request message, a received_auth_request event should be dispatched, and the FSM should count the SRES and send a BSSMAP_Auth_Response message to the SUT. The latter one definitely takes more time. If there is an auth_request event in the queue it will delay the processing of all the other events located after it with at least 100 milliseconds. Since there is no rearrangement in the queue an FSM could cause a serious delay for the others.

According to the POSIX threads, we have introduced the notion of virtual threads. Every FSM, their context in the local database, and their events in the queues belong to a virtual thread and our task is to find the best way to schedule them. A thread could hold back many other virtual threads mainly if the scheduler always looked ahead with only one place (only the next event) in the queue. To avoid this case, the scheduler has to look ahead by more events. To measure this anticipation we introduce the Look Ahead factor (LA) as the number of events in the queue that are taken into consideration by the dispatcher. If the LA has a value of two, the dispatcher counts with the first two elements of the event queue, allowing the second event to be served sooner than the first one, if the first event suffered a delay. To measure the efficiency of the scheduling algorithms we also introduce a metric, which is the sum of the delays suffered by all the virtual threads in the test system (see Eq. 7).

$$m_d(t) = \sum_A \text{delay}_a(t) \tag{6}$$

Where

- A is the set of virtual threads,
- $delay_a(t)$ is the delay of virtual thread a ($a \in A$).

The greater value $md(t)$ has, the less optimal the particular algorithm is. We consider this function as our target function to be minimized. We can shift serving a virtual thread if its timer entries in the timer queue allow us to. To measure the flexibility of this shifting we define the notion of Laxity, which indicates the maximal time we can use to shift the processing without causing actual delays:

$$L_{a_x}(t) = t_{a_x} \cdot \text{deadline-t}_{a_x} \cdot \text{eventreceived} \tag{7}$$

Where

- $a_x \in A$ is a particular virtual thread,
- $t_{a_x} \cdot$ deadline is the deadline of processing the next event belonging to thread a_x,
- $t_{a_x} \cdot$ received is the time of reception of that event.

According to this, in our example five FSMs place their timer entries in the timer queue with one second and five other with 100 milliseconds. If an authorization request (Auth_Req) message arrives within 100 milliseconds, it means that the Laxity of

that virtual thread is 900 milliseconds, so we can handle other threads with a smaller Laxity.

The Look Ahead factor determines the number of events taken into account while the Laxity of the particular events suggests some kind of priority. We have extended the test system with a secondary buffer, having a size identical to the look ahead factor. To process an event from the event queue, the Laxity is calculated first and if it is a positive number it is placed in the secondary queue and the next event is taken. If it is zero or negative, there is no other choice but to handle the event.

The events in the secondary queue are ordered by their Laxity, the event with the smallest Laxity is located on the top. The key question is the size of the secondary queue. If it is too small we will not have any advantages, but if it is too large it will increase the overall delay in the system. There are two extreme cases:

- Every event is moved to the secondary queue without processing them, causing the size of the queue and the Look Ahead factor to increment continuously.
- The length of the queue remains zero.

The value of the Look Ahead factor should adapt to the behavior of the SUT, and so, to the size of the secondary queue. Our algorithm dynamically calculates the optimal LA value and controls the length of the queue. If a virtual thread holds the others back, the Look Ahead factor should be increased in greater steps. Since the metric is the sum of the individual delays, the more virtual threads are held up, the steeper the target function will be. In the following we propose a parameter that indicates the growth of the secondary queue (see Eq. 8).

$$k(t) = \frac{d}{dt} m_d(t) \tag{8}$$

3.4 The improved scheduling method

In the beginning, the size of the secondary queue is zero and the events are taken from the event queue. When the first delay occurs, the growth parameter of the queue is evaluated and the Look Ahead factor is increased according to the growth parameter. When the next event is taken, the value of its Laxity is calculated. If it is a positive number, the event is placed in the secondary queue. If the Laxity is zero or negative the event is served and the next one is analyzed. This step is repeated multiple times according to the value of k. Every time before placing a new event into the secondary queue the elements with zero Laxity in the secondary queue are served. By the end of the previous step at least some of the delayed threads will be served, and k will be decremented by one. If the value of Laxity is still a positive number, the process is repeated. In normal circumstances when there is no delay in the system, the first event from the event queue is compared to the event located on the top of the secondary queue. The element with the smallest laxity is served and the other is placed into the secondary queue. The Look Ahead factor is not incremented in this case. When the Laxity of an element in the secondary queue reaches the value of zero, it is served and the length of the secondary queue will be decreased. See Fig. 7 for the algorithm.

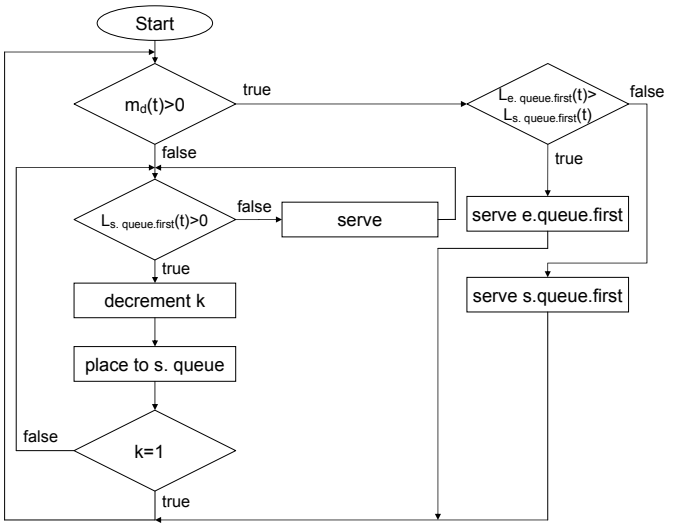

Fig. 7. The scheduling algorithm

3.5 Results

In Section 3 we have presented a new FSM-based approach for load testing. Our aim was to reduce the number of delays introduced by the test system but not by the SUT. For this purpose we have presented an improved architecture and an algorithm, that reduces the number of delays caused by the late processing of an event, that arrived in time to the test system. We have implemented the model in a TTCN-3-based test environment, and we applied it to some theoretical examples. The examples indicated that the algorithm was able to reduce the delay introduced by the test environment significantly.

4 Future Work

In the future we are going to eliminate the constraints of needing to know the exact running times of test components by approximation. As the running time of a test component highly depends on how much the SUT is stressed, a simple approximation for the running time of a test component is the actual running time of the latest test component of the same type. We are going to approximate the required capacities of test components from the TTCN-3 source. We are also going to make our algorithm capable of handling different capacity requirements (e.g. CPU load, memory usage, number of messages per second) by which we expect even better results, and develop methods for approximating the optimal time frame size. We also intend to adjust the scheduling algorithm according to several other load generation algorithms.

References

1 *Framework on Formal Methods in Conformance Testing*, 1997. ITU-T Recommendation Z.500.

2 **Din G, Tolea S, Schieferdecker I**, *Distributed Load Tests with TTCN-3*, TestCom 2006. Proceedings, 2006, pp. 177-196.

3 **Saledo-Sanz S, Xu Y, Yao X**, *Hybrid Meta-heuristics Algorithms for Task Assignment in Heterogenous Computing Systems*, Computers & Operations Research **33** (2006), 820-835.

4 **Salman A, Ahmad I, Al-Madani S**, *Particle Swarm Optimization for Task Assignment Problem*, Microprocessors and Microsystems **26** (2002), 363-371, DOI 10.1016/S0141-9331(02)00053-4.

5 **Ucar B, Aykanat C, Kaya K, Ikinci M**, *Task Assignment in Heterogenous Computing Systems*, Journal of Parallel and Distributed Computing **66** (2006), 32-46.

6 **Garey M R, Graham R L**, *An Analysis of Some Packing Algorithms*, Combinatorial Algorithms, New York: Algorithmics Press (1973), 39-47.

7 *Digital Cellular Telecommunications System (Phase 2+) Mobile Switching Centre - Base Station System (MSC-BSS) interface Layer 3 specification 3GPP TS 48.008 version 7.12.0 Release 7*, 2008.

8 **Wagner F**, *Modeling Software with Finite State Machines: A Practical Approach*, Auerbach Publications, 2006.

9 **Audsley N, Burns A**, *Real-Time System Scheduling*, Vol. 134, University of York - Department of Computer Science Report, YCS, 1990.

10 **Chetto H, Chetto M**, *Some Results of the Earliest Deadline Scheduling Algorithm*, IEEE Transactions Software Engineering **15** (October 1989), no. 10, 1261-1269.

11 *The Testing and Test Control Notation Language, version 3; Part 1: TTCN-3 Core Language*, ETSI, 2005.