

# How to Handle CPU Bound Systems: A Specialization of Dynamic Performance Stubs to CPU Stubs\*

Peter Trapp      Christian Facchi  
University of Applied Sciences  
Esplanade 10  
85049 Ingolstadt  
-Germany-  
{peter.trapp, christian.facchi}@fh-ingolstadt.de

## Abstract

*Dynamic performance stubs provide a framework for the simulation of the performance behavior of software modules and functions. They can be used to realize a cost-benefit analysis of the gain from performance optimization and therefore, for a gain oriented improvement. It is also possible to identify "hidden" bottlenecks and the most relevant optimization candidates. This paper classifies several types of stubbing possibilities and evaluates the CPU Stubs more in detail which can be used to optimize CPU bound modules or functions.*

**Key words:** performance improvements, simulated optimization, stubs, performance modeling

## 1 Introduction

*Dynamic performance stubs* have been introduced in [TRAP2007]. They can be used for a "hidden bottleneck" detection. Knowing the level of optimization potential, a cost-benefit analysis can be done, too. This leads to more gain-oriented performance optimizations. The idea beyond *dynamic performance stubs* is a combination of performance improvements [JAIN1991, FORT2003, LILJ2000, GUNT1998] of already existing modules or functions and the stubbing mechanism from software testing [BERT2005, LIGG2002, SOMM2001]. The performance behavior of the component under study (CUS) will be determined and replaced by a software stub. This stub can be used to simulate different performance behaviors. The optimization expert can use these setups to analyze the performance of the system under test (SUT). This procedure relates to stubbing a single software unit and hence it will be called "local". Therefore a "local stub" has to be build. The *performance simulation functions* (PSF) can also be used to change the behavior of the complete sys-

tem. Therefore a software module has to be created which interacts "global" in the sense of influencing the whole system instead of only one software unit. This stub will be called "global stub".

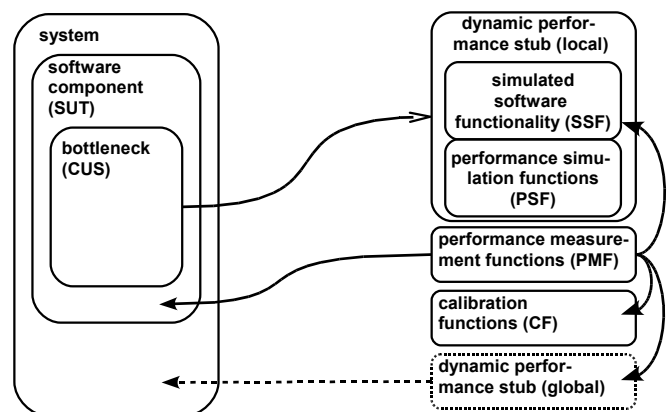


Figure 1: Interactions of "Dynamic Performance Stubs"

Figure 1 sketches the design and the interaction between a real system and the *dynamic performance stubs*. The lined arrow describes a replacement. Filled arrowheads describe the extension of a unit by this fea-

\*This paper has been presented at CMG 2008. <http://www.cmg.org/conference/cmg2008/>

ture and the dashed block provides an additional functionality to the *dynamic performance stub* and will not really stub a software unit.

The framework of the *dynamic performance stub* consists of the following parts which are presented in Figure 1:

- Simulated Software Functionality (SSF)
 

The *simulated software functionality* is used to simulate the functional behavior of the CUS in order to provide proper system behavior.
- Performance Simulation Functions (PSF)
 

*Performance simulation functions* provide the possibilities to simulate the performance behavior of the replaced CUS. They are divided into four categories:

  - CPU
  - Memory
  - I/O
  - Network
- Performance Measurement Functions (PMF)
 

To provide a basic set of evaluation possibilities the *performance measurement functions* can be used. They are mainly glue/wrapper functions for the measurement functions provided by the system.
- Calibration Functions (CF)
 

In order to provide trustworthy results, the stubs have to be adjusted to the dedicated system. This can be done using the *calibration functions*.

The reader is referred to [TRAP2007] for more detailed information on *dynamic performance stubs*.

This paper will shortly classify the different types of *performance simulation functions* and will evaluate *CPU Stubs*, which simulate the performance behavior regarding CPU usage, more in detail. In Section 4 the classification of the stubs will be applied to the *CPU Stubs* and the usability of the *CPU Stubs* will be experimentally evaluated. A methodology for using *CPU Stubs* is also included. Future work and a discussion will conclude this paper.

## 2 Classification of Performance Simulation Functions

There are two possibilities to change the performance behavior of the system using *performance simulation functions*. The first is to use them to replace a software module or function. They are called *local dynamic performance stubs* (see Section 1).

Another possibility to use *dynamic performance stubs* in a more global manner. Therefore, an additional software module can be inserted into the system which changes the resource utilization by introducing additional load. With this method, the whole system will primarily be influenced and therefore it is called *global dynamic performance stubs*. As an example, a module can be added which only uses the *CPU* functionalities of the *performance simulation functions*.

The two main categories can furthermore be separated into two subcategories<sup>1</sup>. The following description relates to the *local dynamic performance stubs*:

**system influencing:** the resource used by the stub and will not be available for further use in the system.

**system non-influencing:** only the SUT will be influenced by the simulation. The resource itself is available to the system.

A classification of the *global dynamic performance stubs* into *system influencing* and *system non-influencing* does not make much sense. The *system non-influencing global dynamic performance stub* would not interact with the system nor with a software module directly. Therefore, all *global dynamic performance stubs* interact *system influencing* without any further notice. The *global dynamic performance stubs* can be categorized as follows:

**open loop:** tries to consume a specified level of system resource.

**closed loop:** tries to adjust to a specified level of system resource usage.

Figure 2 provides an overview of the previous given classification for the *performance simulation functions*.

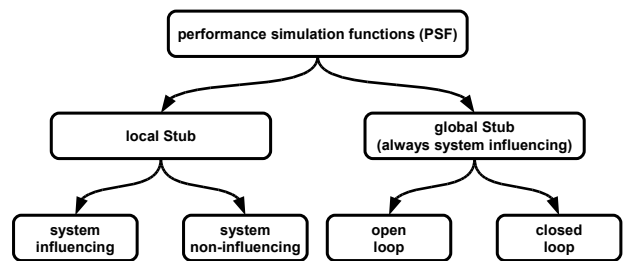


Figure 2: Classification of the *performance simulation functions*

In the following section, this classification will be applied to the *CPU* category of the *performance simulation functions*. They can be used to create a *CPU Stub*, which targets to simulating the timing and cpu behavior of the CUS.

<sup>1</sup>This classification differs slightly from the original classification done in [TRAP2007] to minimize confusion.

### 3 CPU Stubs

An active CPU core can only be running or waiting. If the CPU is waiting it will not execute any other process than the idle. Therefore, the idle state does not have to be simulated.

A process can either use the CPU or can be blocked. If the process is not blocked it will use 100% of the CPU while it is running. In order to simulate the performance behavior of a process the two states “running” and “blocked” have to be simulated. The *CPU-PSF* can be used to simulate the timing behavior of software modules. This mainly relates to the cycles which are used by the software either for working or waiting.

#### 3.1 CPU-PSF

In order to properly simulate the CPU performance behavior of the application the classification from Section 2 is applied to the *CPU-PSF*:

**system influencing:** relates to the time spent in the application while it is scheduled. Here no other module or function can use the CPU.

**system non-influencing:** relates to the time while the application is blocked due to any event. The application is not scheduled and therefore, the CPU can be used.

An example implementation of a *system influencing* and *system non-influencing* realization of the *PSF* simulating the CPU is given below.

#### Example for a system non-influencing CPU-PSF

The simulation of the *system non-influencing CPU-PSF*, which means that the real process is blocked or waiting for an event, can be handled easily by letting it sleep for the desired time.

**Implementation** For instance in UNIX environments the “usleep()” function can be used. For details see “unistd.h”.

#### Example for a system influencing CPU-PSF

The *system influencing* stubbing of CPU cycles can be realized using “no operation” (NOP) so that only the resource CPU will be used by the *PSF*. They have to be executed in a loop in order to reach the desired time consumption. Additionally, the NOPs and also the loop have to be protected against compiler optimizations.

Using this method has some advantages and restrictions. On the one hand, it is easy to implement and calibrate. Moreover, it is mainly architecture independent and only slightly modifies other parts of the system. Almost all needed time values can be simulated by this

implementation. They can range from some nano seconds to several minutes or more (see Section 4.1).

On the other hand, the setup of the *global stubs*, which use the CPU functionality, can lead to undefined results if the CPU highly uses the CPU frequency scaling possibility. In contrast to the “waiting” time, the duration needed for processing a single cycle depends on the actual CPU frequency. If the CPU uses the frequency scaling feature the ratio between “waiting” and “working” will not be constant. Therefore, errors will be introduced in the test results if the frequency of the CPU is changing.

**Implementation** An example implementation can be seen in Listing 1. The function “useCycles” takes a value standing for the processor cycles working for “usec” as input. The “TIME” constant is defined within the *dynamic performance stub* and has been evaluated using the *calibration functions*. Here each iteration will exactly consume 1 $\mu$ s while looping around the empty statement “;”. The same approach is taken inside of the Linux kernel. Here the “BogoMIPS” [LOVE2005, BOVE2005] value will be evaluated while booting. The result is stored in the “loops\_per\_jiffies” variable and used for small delays, e.g. within the *ndelay* function.

```
void useCycles (long usec)
{
    long i;
    for (i = 0; i < usec * TIME; i++)
    {
        ;
    }
}
```

Listing 1: Example implementation of a *system influencing CPU-PSF*

**Discussion** In [TSAF2005, TSAF2007] a problem with simulating a dedicated amount of time with do-nothing loops is described. Despite the problems seen, there are big differences in the approaches. The procedure is targeting the area of bulk-synchronous parallel jobs which are realized as do-nothing loops. The focus is to optimally utilize each of the included processors. So the processes always try to run, ignoring the amount of time needed for the operating system per processor. As soon as the operating system has something to do the userspace application will be scheduled out and the total execution time will be delayed. Our loops, however, will be calibrated in an otherwise idle system with enough time for the operating system (see also Section 3.3). As experimentally proved in Section 4.1 in our environment the execution time of a process can be simulated with a do-nothing loop, predictably in

contrast to [TSAF2005, TSAF2007]. Additionally, because of the fact that such a loop has a defined number of instructions these loops can be used to simulate the timing behavior of processes.

### 3.2 CPU-PMF

The *CPU performance measurement functions* are mainly functions already provided by the system for the reason to measure time. More specific the time can be measured at several points in the system, e.g. using a separated process for measuring or directly within the CUS or dynamic stub. Also different types of time can be measured. First of all, there is the total time spent in the system. Architectures normally provide two devices for time keeping [LOVE2005]. The first is the RTC (real-time clock) which is system independent and keeps track of the absolute time. It is normally used by the kernel to initialize the wall clock time but can also be read by other applications [BOVE2005]. The second is the TSC (time stamp counter). It is updated with each tick of the system and can be used to measure the total time passed in the system. These are possibilities measure to the total time accurately. For a more granular and portable wall clock time measurement the POSIX standard <sup>2</sup> specifies several functions such as the “gettimeofday()” (sys/time.h; time.h) or the “time()” (time.h).

Linux also provides the possibility to measure the time used by the dedicated process. For this purpose the “clock()” (time.h) or the “times()” (sys/times.h) function can be used. Also more detailed information can be gained by using the “getrusage()” (sys/time.h; sys/resource.h).

According to the needs of the performance analysis one or more of the above mentioned measurement functions fits more or less well. Therefore, the most suitable function shall be chosen. In our environment we used almost all of the above mentioned measurement possibilities to validate our approaches and to improve the measurement results.

Beside the time measurement, it is often useful to measure the utilization of the CPU usage. There are several dedicated applications for a system wide measurement, just to name two: “top” and “vmstat”. To measure the CPU utilization from inside of a module or function the “proc” - filesystem can be used. Here the “stat”-file provides a global view, e.g. of the CPU utilization. To measure only self-used information within the process, the “stat”-file which is located in /proc/(pid)/stat can be used.

<sup>2</sup><http://www.pasc.org/plato/>

### 3.3 CPU-CF

The *CPU calibration functions* as already discussed in [TRAP2007] are used to adjust the *PSF* to a dedicated hardware architecture.

Several steps have to be taken for setting up the *CPU-PSF* to the system. In common the setup consists of mainly two parts, calibration and validation which are both included in our approach.

The *system non-influencing CPU-PSF*, as described in Section 3.1 are only using the system internal waiting functions. Because these functions are already delivered by the operating system, nothing has to be adjusted here.

```
long long int calibrateLoop (long nLoops)
{
    long i;
    long long int beforeTSC;
    beforeTSC=readTSC ();
    for (i = 0; i < nLoops; i++)
    {
        ;
    }
    return ( readTSC () – beforeTSC );
}
```

Listing 2: Example implementation of a *system influencing CPU-CF*

For the *system influencing CPU-PSF* the number of loops for a predetermined duration has to be evaluated, e.g. the value of the “TIME” constant in Listing 1. This can be done as shown in Listing 2. This function has to be called, changing the “number of loops” value, until it returns the desired time value with the needed precision. Of course, the overhead for the “readTSC()” function call has been determined and subtracted. A similar approach has also been taken to calculate the “bogoMIPS” value<sup>3</sup>.

In order to ensure the quality of the result the execution of the function shall not be interrupted. This can be achieved by the following preconditions:

- Executed with high priority
- Executed on an idle system
- Using a tickless kernel
- Adding a short “system recovery” time between each execution. So other processes can use the CPU without disturbing the calibration function.

Additionally, it has to be ensured that no interrupt took place, e.g. using the getrusage() function call.

After determining the “number of loops” the result has to be validated. The first step is to validate the proper working for the predefined amount of time by executing

<sup>3</sup>Linux kernel source init/calibrate.c

the “calibrateLoop()” function statistically sufficiently often with “number of loops”. If the results are sound the second evaluation can take place. Here, the proper working of the *system influencing CPU-PSF* for the time range, which is necessary to simulate the timing behavior of the bottleneck, has to be validated. Therefore, the “number of loops” has to be recalculated using the rule of proportion and the measurements has to be done again. To improve the accuracy of the results the steps as described above can be taken. The quality of the results can be evaluated by applying the linear regression method to the measured values and calculating a confidence interval.

## 4 Validation

As test environment the test PC of the Nokia Siemens Networks ngRNC (next generation Radio Network Controller) is used to validate the measurements. It hosts a 2.8 GHz Intel Xeon central processing unit with hyper-threading disabled. The operating system is a standard Linux running a 2.6.22 kernel. The kernel was build tickless with the high resolution timers enabled.

As shown in Section 3.1 a “do-nothing loop” will be taken to simulate used CPU cycles. This approach is also taken in the Linux kernel [LOVE2005].

All measurements are done on an otherwise idle system, where only core processes are running. We also disabled the CPU frequency scaling feature in all our test runs. We have chosen the GCC (GNU Compiler Collection)<sup>4</sup> as compiler for our dynamic stubs.

We used a deterministic load if we needed to test our *dynamic performance stubs* in a situation where the system was not idle. This ensures the reproducibility of the results. Typical workloads are either a process, which uses a dedicated amount of the CPU or the process itself running multiple times. Despite that all examples presented in this paper are realized as global stubs, the results are also applicable to stubs in general. The authors refer to [TRAP2007] for more information.

### 4.1 Simulation of the time

This section shows that the time can be simulated with high precision. Therefore, the methodology for calibrating the *CPU-PSF* as described in Section 3.3 has been used. Our goals were to validate:

1. The simulation of different time intervals within a big range.
2. The accuracy of the simulated time interval.
3. The usability in different software development environments.

<sup>4</sup><http://gcc.gnu.org>

First of all the “number of loops” has been determined for one second. This is the base for all measurements which has been done. The main loop for each run can be seen in Listing 3, where “workInit” was initially set to the “number of loops” needed to simulate the highest simulated time value.

```

for ( i = 0 ; i < samples; i++)
{
    work = workInit * i;
    beforeTSC=readTSC();
    for ( j = 0; j < work; j++)
        ;
    deltaTSC[i] = readTSC() - beforeTSC;
    usleep(1000);
}

```

Listing 3: Simulation of a duration

The graph in Figure 3 shows linear increasing of the “number of loops” which was initially set to simulate one second. The y-axis plots the needed time used inside of the application as TSC value. On the x-axis the actual number of samples can be seen. In Figure 3 each sample took 1ms.

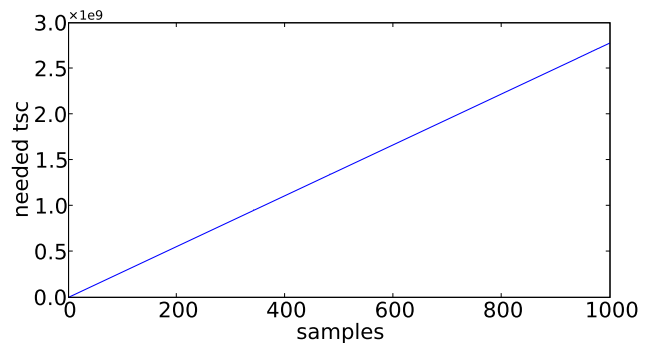


Figure 3: Simulation of time ranges

In order to prove the accuracy of the results, the overhead for measuring was subtracted from the measured TSC value. Additionally, the measured TSC value has been divided by the sample number and the value for zero loop iterations has been ignored. This results have been used to analyze the minimum, mean and maximum values and to calculate the square root coefficient of variation (sqd coeff of var). The results of the evaluation of the values from Figure 3 are summarized in the second row of Table 1. Additionally, the table shows further evaluation done as described above. Here the time ranges from 0 to 1μs and from 1 to 10 seconds.

Since we have shown that our approach of simulating time can be used to create *CPU Stubs*, we wanted to ensure that it can also used in different software development settings. Therefore, we firstly changed the compiler and secondly the operating system and architecture<sup>5</sup>.

<sup>5</sup>An Intel® IA-32 CPU has been used.

range [s]	# loops	samples	min	mean	max	sqd coeff of var
[0;1] $\mu$	[0;398]	100	26.1	34.15	82.83	0.10794
[0;1]	[0;462356181]	1000	2777453	2779004	2793478	0.000000055
[0;10]	[0;4294963200]	1000	26861410	26879512	27023427	0.000000169

Table 1: Trustworthiness of the time simulation

In Table 2 the results of this evaluation are shown. The first row is taken from Table 1. This line will be compared to measurements, where the binary has been build with the ICC (Intel C++ Compiler)<sup>6</sup>. The results are almost the same and do not show any surprising values.

The values of the second test, which are presented in the last row of Table 2, can not directly be compared to the other values. Here, an Intel Pentium M with 1.6GHz and Microsoft Windows XP Professional<sup>7</sup> has been used. The results show some slightly worse behavior, which can easily be explained by the change of the operating system. The used operating system has been running more “core”-processes and has not been optimized for this special task. Where else, the Linux Kernel has been explicitly build and optimized, no changes were applied to the kernel of Microsoft Windows XP Professional. Nevertheless, the values are still fine and our methodology can also be applied in this environment.

**Conclusion** We have experimentally proved that the time and therefore the usage of the processor can be simulated by “do-nothing loops”. Here, the values for simulating time spans from several nano seconds to a couple of seconds and possibly more. Also the accuracy of the simulated time is sophisticated enough to use the “do-nothing loop” for simulating the CPU behavior of processes. Additionally, a different software development setup do not highly influence our approach as proved above.

## 4.2 Open Loop

“Open Loop” in this context means, that the stub should consume a dedicated amount of CPU usage for a definable time slice, e.g. 50% for the next 5 minutes. This can be used to simulate the performance behavior of a CPU bound bottleneck in order to replace it properly as part of the *CPU-PSF*. The “open loop” can be realised by a period with many sleeping and working faces by turns. So no feedback loop is in place.

Figure 4 shows 20 single runs starting from 5% CPU utilisation with a 5% increase per step. The y-axis

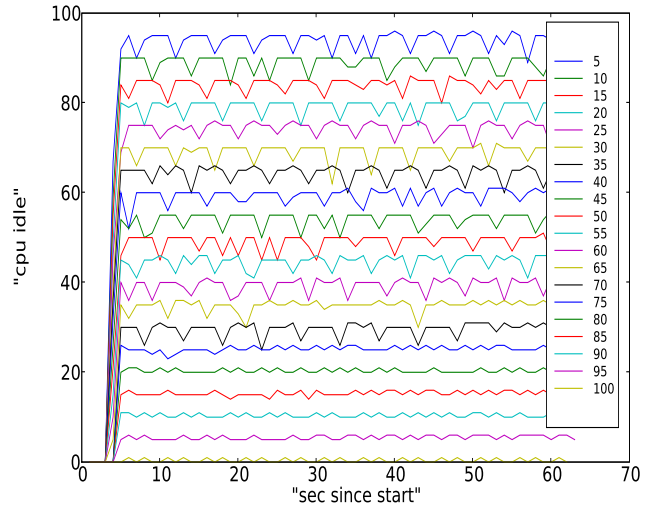


Figure 4: Global open loop CPU stub using a dedicated amount of time

shows the idle value of the CPU and the x-axis provides the seconds since the test run was started. Each run took approximately 64 seconds where the first 4-5 seconds were used to setup the working and waiting periods including the calibration of the *CPU-PSF* to the system. The graphs have been measured in the system using a standard performance measurement tool. The tracing interval was set to one second which explains the two starting points of the simulation.

As can be seen in the Figure 4, the stub uses a dedicated amount of the CPU. Due to the open loop it will not control the system utilization and therefore it will not adjust its values. The peaks, which can be seen in the graphs, can be explained by the work which has to be done by the system.

This behavior can be used for a *local CPU Stub* to simulate a process which periodically works and then waits for an event. Another possibility is to use it as a global stub to increase the CPU utilization or to transfer a system from non- to CPU bound.

The time used for the open loop was calibrated to 1 second. This means that there were always “huge” blocks of “waiting” and “working”, e.g. for 50% there is 0.5s working and 0.5s waiting.

This behavior mostly appears in a range around the middle percentages and is only hardly true for real applications.

For smaller percentages the behavior of “1 second”

<sup>6</sup>Intel<sup>®</sup> and executed in the same environment. The ICC can be found at <http://www.intel.com/cd/software/products/asmo-na/eng/compilers/284264.htm>

<sup>7</sup>Microsoft<sup>®</sup> Windows XP Professional<sup>®</sup>

<sup>8</sup>Intel<sup>®</sup> Pentium M (IA-32 Architecture)

range [s]	# loops	samples	min	mean	max	sqd coeff of var	setup
[0;1]	[0;462356181]	1000	2777453	2779004	2793478	0.000000055	Linux & GCC
[0;1]	[0;462819751]	1000	2777064	2778785	2793469	0.000000032	Linux & ICC
[0;1]	[0;225197530]	1000	1286750	1578846	1583827	0.0000518	Windows & GCC <sup>8</sup>

Table 2: Portability of the simulation of time

calibrated stubs can be taken for simulating “special” kind of applications, e.g. I/O bound, where the process shortly works and then waits for further input. For higher percentages the stub can simulate, e.g. number crunching applications. Here large “working periods” are shortly interrupted by small “waiting periods”.

We have also evaluated several approaches to scale down the “big blocks”. Therefore, we have redone the measurements with smaller blocks of “waiting” and “working”.

**Scaling** The size of the finest granularity depends on the smallest time which can be simulated. As shown in Section 4.1, the smallest “working” time is bound close to the length of a couple of cycles. Despite the fine granularity of the “working” time, the smallest reliable “waiting” time for userspace applications is 1 $\mu$  second. Therefore, the “waiting” period is the bottleneck for scaling down big blocks into smaller blocks. Our approach is to take the 1 $\mu$ s as the total waiting time for each loop. This means the 1 $\mu$ s simulates the 100%-x%, where x means “supposed working percent”. Therefore, the time needed for the “x working percent” has to be added to the 1 $\mu$ s. This approach works really well if the percentage is small. If the “working percent” increases the additional time will take the lion’s share of the total time, e.g. for 98% working the total time is close to 100 \* the smallest possible (here 1 $\mu$ s).

Therefore, we also thought about another approach. This is the least common denominator which evaluates the smallest common denominator of 100% and the desired x% value. For the example above, the total time to simulate 98% comes down to 50 \* the smallest possible time. Additionally, we also allowed some inaccuracy to the simulated x percentage value. Instead of only allowing exactly x% we extended the value to some user given interval, e.g. to simulate 67% the value has to be between [66.5;67.5]. For that reason our algorithm matches the smallest possible denominator to create a percentage which is in that range. For example, our algorithm returns for the input of 67% a numerator of 2 and a denominator of 3. Compared to the least common denominator, the algorithm brings the total time down from 100 \* smallest time to 3 \* the smallest time.

For the simulation of different kinds of processes also multiples of the least common denominator can be used.

### 4.3 Closed Loop

The closed loop tries to adjust the CPU utilization to an user specified percentage by using a feedback loop. We experimentally implemented a time series analysis algorithm. It is based on a simple moving average with evaluation of the last five values of the original CPU load. The sample rate is set to 4Hz.

We used three different workload types for proving the concept. The first execution was done in an otherwise idle system and the second with a constant CPU utilization where the original utilization was below the supposed percentage. The results are similar as the results of the “open loop”. This is nothing special and therefore not further discussed in this paper.

The third evaluation was done with a variable CPU utilization. The original load signal is presented in Figure 5. The figure shows on the x-axis the iteration of the evaluation step, the y-axis shows the utilization of the CPU. The system load signal ranges from almost idle with some small peaks to shortly fully utilization and then varies around the to-be-adjusted value, which was in this case 50%. The figure also presents the controlled process variable which is the second graph in the figure and called adjusted.

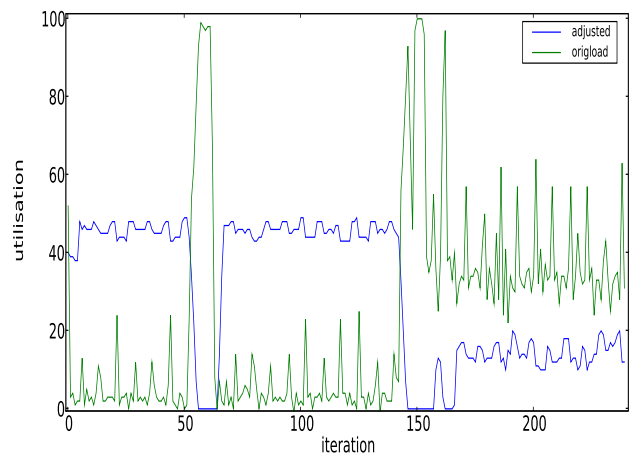


Figure 5: System signal and control signal for the closed loop algorithm

The real utilization was measured in the system and can be seen in Figure 6. The axes of this figure are the same as the figure above except of the sample rate which was set to 1 second. The load in the system varies as supposed but the average is 52.40% which is

quite close to the predefined 50%.

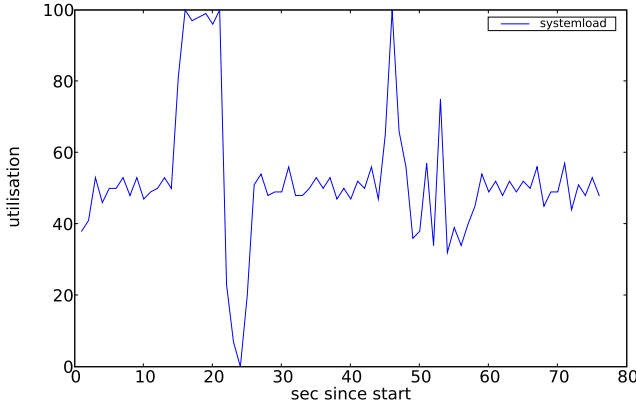


Figure 6: Total system load measured in the system while closed loop algorithm was running

**Conclusion** We have proved that the CPU utilization can be adjusted to a defined value as long as the original utilization of the CPU is lower than the supposed value.

This evaluation should only present the functional capability. A more theoretically and sophisticated methodology will be done as next steps. It will probably be based on the host load prediction system as described in [DIND2000, DIND1999] and the Box-Jenkins methodology [BOX1994].

The closed loop can be used to adjust the utilization of the CPU to a defined value. This can be helpful for performance tests if the system should be tested under high load. Additionally, it can be used to deterministically create overload to validate the proper working and performance of overload routines under “real conditions” or to constantly influence other processes in the system.

## 5 Towards a Methodology

So far only the content of a toolbox has been presented, but now the first steps towards a methodology on using CPU stubs for performance improvement is given. So this chapter can be seen as first version of a how to.

1. Determine CUS: As a first step of any optimization the timing behavior of the system in total has to be examined. Then one component of the system (CUS) has to be chosen, which seems to be a bottleneck [TRAP2007, JAIN1991]. Then several performance measurements have to be done until the results seem to be deterministic within a given confidence interval. The following values shall be deterministic:

$t^{CUS}$  the time which will be used in the CUS

$t^{SUT}$  a performance indicator of the module or function. This value will be measured to validate the systems performance in total. Often it is the time used for the completion of a service.

$t^{CUS_{busy}}$  the time which will be used in the CUS by the CPU. This can be done by measuring the wall clock time and the time spent executing (e.g. TSC [ETSI2000] and getrusage). The result will be the ratio between working and waiting:  $\frac{t^{CUS_{busy}}}{t^{CUS}}$ . This is only a first approximation. It may work but not essential. Please see Section “Conclusion and Future Work”.

This step has to be done a couple of times in order to get some statistical distribution of the required values. If there is only a small variation we can go further.

2. Create CPU Stubs. After creation of the functional stub, the following CPU stubs will be generated:

**Idle Stub:** A stub has to be created, in which no CPU will be used. So only a sleep will add load to the CPU, as described in Section 3.1. Then several measurements have to be done until the results are deterministic. So we will have  $t_{idle}^{CUS}$  for the stubs used time and  $t_{idle}^{SUT}$  for the time in the module or function.

**Busy Stub:** A stub has to be created, in which the CPU will be used. So a busy loop will add load to the CPU, as described in Section 3.1. Then several measurements have to be done until the results are deterministic. So we will have  $t_{busy}^{CUS}$  for the stubs used time and  $t_{busy}^{SUT}$  for the time in the module or function.

**Mixed Stub:** A combined stub has to be created (see also Section 4.2), in which the CPU will be used according to ratio seen in Step 1:

$$\frac{t^{CUS_{busy}}}{t^{CUS}}$$

So realistic load will be added to the CPU, as described in Section 4.2. Then several measurements have to be done until the results are deterministic. So we will have  $t_{realistic}^{CUS}$  for the stubs used time and  $t_{realistic}^{SUT}$  for the time in the module or function.

3. The measurement results have to be validated. The following possibilities can be seen:

$\neg(t^{CUS} = t_{idle}^{CUS} = t_{busy}^{CUS} = t_{realistic}^{CUS})$  Then the replacement of the CUS by a stub does not work properly, because as a requirement of the stub all values should be the same. There are only smaller statistical variations allowed. In that case, go back to Step 1 or a change of the measurement is necessary.

$t_{busy}^{SUT} < t_{idle}^{SUT}$  That is a strange result, because the complete system will be faster if the CPU load has been increased. So the measurement seems not to be appropriate and a detailed look on the performance is necessary. So go back to Step 1.

$t_{busy}^{SUT} = t_{idle}^{SUT} = t_{realistic}^{SUT}$  So the overall system is mainly not CPU bound, because a reduction of the CPU load does not lead to a performance improvement. However, if the CUS will be optimized regarding the execution time the performance of the complete system can be improved, because of change of the timing behavior.

$t_{busy}^{SUT} > t_{realistic}^{SUT} > t_{idle}^{SUT}$  Then any optimization of CUS has two effects:

- Performance gain by reduction of execution time.
- Performance gain by reducing the CPU load as an additional acceleration.

4. Now a first estimation of the optimization of the CUS has to be done. Normally the author of this component has rough estimations of performance improvement. Then a stub has to be taken, where this time will be used. Therefore the stub to yield  $t_{realistic}^{CUS}$  can be taken. The determined time will be called  $t_{improved}^{SUT}$ . This is a measurement for the improved system behavior after optimization.

5. After optimization determine new CUS and go to Step 1.

Additional to Step 4) some more measurements can be done. Now let *improvement\_ratio* be the factor of improvement which will be realized by the stub. Then  $t^{SUT}(improvement\_ratio)$  denotes the complete time when a stub with the desired ratio is in place. Now different measurement can be done:

- If  $t^{SUT}(0) = t^{SUT}(1)$ , then the CUS is not the bottleneck, because any optimization effort does not lead to a performance gain.
- A detailed improvement analysis can be done, e.g. make several measurements starting with *improvement\_ratio* = 0 and increase to 1 using a step size of 0.1.
- Make a cost estimation of the improvement function. Make a detailed cost-benefit analysis to determine the ideal optimization factor. This will lead to a solid basis for a performance optimization.

## 6 Conclusion and Future Work

In this paper a specialization of *Dynamic Performance Stubs* to *CPU Stubs* has been realized. So the im-

plementation of generic *Performance Simulation Functions (PSF)* have been shown. These *PSF* can be used on arbitrary operating systems, because they are based on standard functionality, which is available on every operating system. The validation of our approach has been realized only on a Linux system, where we have shown that the *PSF* for *CPU Stubs* work properly.

We have introduced *global* - and *local CPU stubs*. With *global CPU stubs*, which can be seen as a load generator, we have shown that the desired load can be generated. Despite the fact that the “closed loop” algorithm, which realizes an adaptive load generation, is working properly a more theoretical foundation in the area of load prediction has to be done in future.

The implementation of *CPU-PSF* can handle two different behaviors: *busy*, where the CPU will be used, and *idle*, where the CPU will not be used. In a component to be optimized both behaviors will happen in a mixed way, because normally a process will calculate some results, where the CPU is used, and has to wait for some external events, where the CPU can be used by other processes. However, there will be some further work necessary to determine the exact ratio of the component, because currently only a rough estimation is done. Additionally, the time spent in a process is normally subdivided into user - and system time. *CPU Stubs* should be extended to simulate these different times, because that will influence the performance behavior of the system.

Also *CPU Stubs* do not consider the functionality of CPU frequency scaling and the support of multiprocessor architectures. These problems should be handled in the next steps.

During the validation it has been shown that the performance behavior of a stub can be modeled in high accuracy, despite the results mentioned in [TSAF2005, TSAF2007]. In our approach we have restricted the system in a way that the system does not disturb the desired performance behavior. These restrictions have not been in contrast to our validation system, which has been an example of a productive system.

A classification of different types of CPU bound processes has to be done. This will help the performance analyzing person by creating *CPU Stubs*. Additionally a methodology on how to create functional stubs will be carried out.

Based on the *CPU Stubs* a methodology for performance optimization has been defined. This can be seen as a first version of a how to, which can be used for most optimization problems in the area of CPU bound modules or functions. The steps towards a methodology, as presented above, mainly works for stubbing deterministic functions or modules. A possibility to stub also non-deterministic functions has to be elaborated as well as the evaluation techniques, e.g. statistical evaluations of the performance measurement results.

Case Studies on the results and methodologies will be carried out.

## 7 Acknowledgment

This research is granted by Nokia Siemens Networks. The authors would like to thank the UMTS group, especially Rudolf Bauer and Helmut Voggenauer as representatives for the excellent support and contributions to this research project.

We would also like to thank the Software Technology Research Laboratory from the De Montfort University for providing the appropriate environment for research.

## References

- [BOVE2005] D. P. Bovet and M. Cesati. *Understanding the LINUX KERNEL*. O'Reilly, 3 edition, 2005.
- [BOX1994] George E. P. Box, Gwilym M. Jenkins, and Gregory C. Reinsele. *Time series analysis: forecasting and control*. Prentice Hall, Englewood Cliff/N.J., 3 edition, 1994.
- [BERT2005] A. Bertolino and E. Marchetti. *Software Engineering: The Development Process - A Brief Essay on Software Testing*, volume 1, chapter 7, pages 393–411. John Wiley & Sons, Inc., 3 edition, 2005.
- [DIND1999] Peter A. Dinda. The statistical properties of host load. *IOS Press*, 7(3-4):211–229, 1999.
- [DIND2000] Peter A. Dinda and David R. O'Hallaron. Host load prediction using linear models. *Cluster Computing*, 3(4):265–280, 2000.
- [ETSI2000] Y. Etsion and D. Feitelson. Time stamp counters library - measurements with nano seconds resolution, 2000.
- [FORT2003] P. J. Fortier and H. E. Michel. *Computer Systems Performance Evaluation and Prediction*, volume 1. Digital Press, Burlington, 2003.
- [GUNT1998] N. H. Gunther. *The practical performance analyst*. McGraw-Hill Education, 1998.
- [JAIN1991] R. Jain. *The art of computer systems performance analysis*. Wiley and sons, Inc., 1991.
- [LIGG2002] P. Liggesmeyer. *Software-Qualität : Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag GmbH, Berlin, 2002.
- [LILJ2000] D. J. Lilja. *Measuring Computer Performance: a practitioner's guide*. Cambridge University Press, New York, 2000.
- [LOVE2005] Robert Love. *Linux-Kernel-Handbuch*. Addison-Wesley Verlag, 2005.
- [SOMM2001] I. Sommerville. *Software Engineering*. Addison-Wesley, 6 edition, 2001. german redaction.
- [TSAF2005] Dan Tsafir, Yoav Etsion, Dror G. Feitelson, and Scott Kirkpatrick. System noise, os clock ticks, and fine-grained parallel applications. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 303–312, New York, NY, USA, 2005. ACM.
- [TRAP2007] P. Trapp and C. Facchi. Performance Improvement Using Dynamic Performance Stubs. Technical Report 14, Fachhochschule Ingolstadt, August 2007.
- [TSAF2007] Dan Tsafir. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *ExpCS '07: Proceedings of the 2007 workshop on Experimental computer science*, page 4, New York, NY, USA, 2007. ACM.