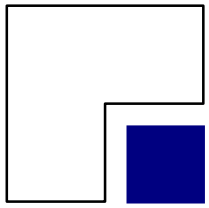
Fachhochschule
Ingolstadt
University of
Applied Sciences

Arbeitsberichte
Working Papers

Kompetenz schafft Zukunft
Creating competence for the future

Performance Improvement Using Dynamic Performance Stubs

Dipl.-Ing. (FH) Peter Trapp
Prof. Dr. Christian Facchi



Fachhochschule
Ingolstadt

University of
Applied Sciences

Arbeitsberichte
Working Papers

Performance Improvement Using Dynamic Performance Stubs

Dipl.-Ing. (FH) Peter Trapp
Prof. Dr. Christian Facchi

Heft Nr. 14 aus der Reihe
"Arbeitsberichte - Working Papers"
ISSN 1612-6483

Ingolstadt, im August 2007

Abstract

Dynamic Performance Stubs support performance improvement. They can be used to identify "hidden" bottlenecks and also to provide better estimations of the gain from performance improvement. The idea behind these stubbing mechanism will be described, the core concept will be explained and a closer look on the possibility of creating stubs will be given. Furthermore an introduction on "How to use *dynamic performance stubs*" is also included.

1 Introduction

Current performance improvements follow a strict line: analysis - test - improvement - verification. This procedure depends strongly on the experience of the person realizing a performance optimization and can only roughly be used for estimations. One of the problems using this methodology is the cost-benefit analysis of improvement possibilities. The expert has to know the complete system at a very detailed level which is hardly possible within large software projects. Another problem of this approach is that the optimization has to be done without really knowing how much of the module has to be optimized [1]. Is it enough to slightly modify performance-critical functions or is it necessary to find a new algorithm for the complete procedure?

After the optimization of some parts normally other additional performance bottlenecks delay the execution time of the component under study (CUS) [1]. Without knowing all bottlenecks in advance the improvement effort may lead to an unexpectedly small gain.

Most of these problems can be bypassed with the *dynamic performance stub*. It combines performance measurements with the well-known techniques of stubbing used in the implementation and test phase of software development. To develop *dynamic performance stubs* the performance of the already existing software modules (systems under test, SUT) have to be studied at a very detailed level and several bottlenecks have to be identified. This performance behavior can be modeled using the *performance simulation functions* (PSF). The resulting performance stub can be used for modeling different performance behaviors of the replaced unit, for estimating the achievable gain or to identify other bottlenecks.

There are two different research scopes in simulation of the performance behavior. In [2] it is explained how performance of inner loops can be modeled at the instruction level and which effect they have on the memory/-cache performance. Although the possibility of modeling software modules exists, the high degree of granularity of this approach reduces the usability for stubbing whole software functions/modules. In [3] the usage of smart stubs for software analysis of functions and modules which are partly not available yet is described. Therefore the stubs simulate the budget regarding storage and timing resources, which have been estimated, for the to-be-implemented software parts. Also mainly a management point of view for the non-existing software will be taken. However, the *dynamic performance stubs* in our approach will be used for stubbing already implemented and measured software parts in order to find the bounds of the performance improvement within that part. This procedure helps to identify the real gain of the performance improvement without really improving it and additionally shows the next bottleneck. So the cost-benefit analysis for improvement activities can be achieved in a more realistic way, because a proper simulated result is better than a simple estimation.

All results will be verified in practice on a software platform for UMTS delivered by Nokia Siemens Networks.

The rest of the paper is structured in the following way. Section 2 provides a small overview about current performance improvements and the method of performance engineering regarding to performance stubs. Section 3 describes the design of *dynamic performance stubs*. A method of using stubs for performance improvement can be found in Section 4. Sections 5 and 6 handle the advantages as well as problems and restrictions of the performance stubs. Conclusions and future work are described in Section 7.

2 Classification of Dynamic Performance Stubs

The aim of each software performance optimization project is to find bottlenecks and to optimize the corresponding software unit in order to reduce the execution time [4, 5]. A bottleneck is the part of the software with the highest utilization/delay and contributes most

to the used time in total respectively to the throughput of a system. Improving this bottleneck has the strongest effect on the system performance and identifying it is the most challenging part in each performance engineering project [5].

An overview of a performance evaluation study can be seen in the Box 2.2 of [5] or in [6]. Information about tools used in performance improvement studies are described and classified, e.g. in [7] and [8].

Beside of the straightforward software performance optimization procedure there is also a more general approach to ensure the quality of the software relating to the performance. It's called "performance engineering" (see also [9]) and combines several concepts of software engineering and performance improvement disciplines.

Performance improvement using *dynamic performance stubs* contributes to some of the parts of the performance engineering method. The descriptions are taken from [6]:

- Performance Measurement

  Measurements on existing software to verify performance criteria or to identify specific performance problems.

- Performance Tuning

  A discipline to improve slow code in order to reach the performance targets.

- Software Engineering

  A discipline to develop large software systems.

- Software Quality Assurance

  Includes requirements for audits and for software product quality evaluations.

The *dynamic performance stubs* models the performance behavior of software modules or bottlenecks and can be used for measuring and tuning the software. Additionally they can be taken for testing which is a part of the software development process. Therefore it contributes to the "software quality assurance".

3 Dynamic Performance Stubs

As already described, the concept of *dynamic performance stubs* combines the methods of software testing [10, 11, 12] and performance improvements [5, 7, 8, 13]. There are only little differences between stubbing for performance improvements and stubbing for testing reasons. Normally stubs are used for simulating remote systems or for non-existing software modules and functions, as in software testing [14]. In this approach the CUS will be replaced by *dynamic performance stubs* in order to simulate the performance behavior of this software unit as a primary goal. This procedure relates to stubbing a single software unit and hence it will be called "local". The *performance simulation functions* (PSF) can also be used to change the behavior of the complete system. 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.

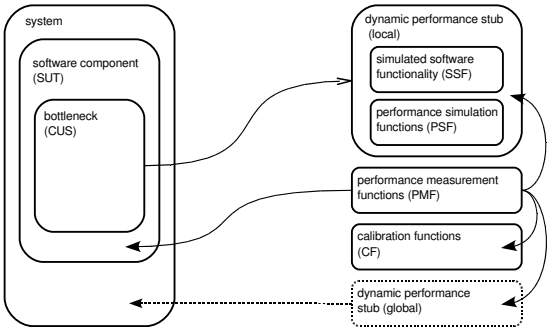


Figure 1: Interactions of "Dynamic Performance Stubs"

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 feature 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)
- performance simulation functions (PSF)
- performance measurement functions (PMF)
- calibration functions (CF)

The *simulated software functionality* (SSF) is additionally implemented code in order to simulate the functional behavior of the already exiting CUS. More information about a possible methodology of stubbing for already existing software modules will be described in Section 3.2.

After creating the functional stub (using the SSF) the performance behavior of the CUS can be modeled using the *performance simulation functions* (PSF). They will provide the possibility to simulate different isolated performance parameters such as the time spent in the component. The functions can be combined in order to simulate the "real" performance behavior. Please refer to Section 3.3 for a wider view of performance parameters which can be simulated by the *PSF*.

Some basic performance evaluations can be done with the *performance measurement functions* (PMF). They will provide a rough overview of the system behavior, e.g. the number of context switches while the stub has been executed. The *performance measurement functions* only extend the possibilities of the stubs and does not have to be included in the stub. They can also be replaced by more specific or detailed performance measurement tools, such as [15, 16, 17].

Additionally the concept of the *dynamic performance stubs* contains the *calibration functions* (CF). These are important for the initial setup of the stubs to the target, e.g. the time needed for an "empty loop" will be determined in order to setup a realistic behavior. This measurement will be done using the *performance measurement functions*. The *calibration functions* have to be executed once for each hardware system.

3.1 Basic Design Decisions

The major decisions regarding the design and implementation of the *dynamic performance stub* are:

1. The programming language used in the project by Nokia Siemens Networks is C/C++. Therefore the *dynamic performance stubs* have to be written in C. This language has several advantages such as the possibility of inling assembler code if a high optimization level is necessary. Additionally C/C++ can be compiled and used as a binary executable code for the native machine language of the CPU used. This is highly recommended in [18] since the programming language influences the performance of the complete system. Moreover, most of the C/C++ compilers provide different optimization flags, e.g. [19]. However, the methodology of the *dynamic performance stub* is independent to the implementation language and can also be used with interpreted, just-in-time compilation or intermediate code.
2. The simulation functions should be configurable and adjustable to the target. This should be done only once during the setup of the stubs.
3. Some basic performance measurement functions should be available. This is necessary for calibrating the performance simulation properties and for proving that the stubs are working properly.
4. The simulation functions of the performance behavior should be used as a toolset. All functions should be accessible without further need of configuration.

3.2 Simulated Software Functionality (SSF) - Stub Generation

Commonly a stub in software engineering is used as a proxy and provides an adequate replacement for the behavior of the to-be-implemented software modules and functions.

It is mainly used in software testing of distributed systems or in modularized software [12].

Therefore the stubbed function uses the same I/O parameters to simulate the missing procedure and provides only a basic functionality, e.g. it returns the results by reading them from a hash table, does nothing or it may simply write trace messages. From the systems point of view the stub will be included and work just as the real procedure [14].

The idea behind the *simulated software functionality* and the functional stubs is nearly the same. The only difference between stubbing for performance reasons and the generation of "test-stubs" as mentioned is the part of software which will be used. Stubbing for performance reasons means to replace an already existing code whereas stubbing for testing creates basic functionality of non-existing software [12, 14] or they are used for remote systems.

A methodology of stubbing already existing and deterministic software functions can be described as follows. First of all the in- and output values of the CUS have to be identified while tracing them. Therefore proper tracepoints have to be inserted in the source code and then the software has to be re-run with a proper and deterministic test case scenario. Rewrite the CUS using the in- and output values (e.g. using a hash table) and by replacing the time consuming functions where the software normally walks through. Now the stub should be working properly.

3.3 Performance Simulation Functions (PSF)

A *PSF* simulates the non-functional behavior of a function regarding to one aspect of performance. They are divided into four classes according to the performance bounds to which a software can belong. For a more detailed description the reader is refered to [20]. The classes are as follows:

- CPU
- Memory
- I/O
- Network

A fifth category can be thought of is the User [5], in our approach it will be included for simplification in the I/O category of the *PSF*.

Almost each of the categories can be divided into a *globally visible* and a *locally visible* part. These subdivision indicates whether the stub will only have local interactions or if it will affect the whole software system (global). This is needed for classification for the *PSF* and should not be mixed with the *global-* and *local dynamic performance stub* mentioned in Section 3 where the stub can be realized as replacement of a software part or can be used globally, e.g. as a kernel module. A closer look in the categories will be given in the following subsections.

3.3.1 CPU-PSF

This *PSF* relates to the clock cycles used by the software. One part of the *CPU-PSF* is the time spent in the stub, while it is scheduled. This is called *globally visible* because no other module can use the CPU while the stub is running at the same time. In the case of a *locally visible CPU-PSF* the module will be blocked due to a timing event. Here the CPU is free for other processes. An example implementation of a *global-* and *local visible* realization of the *PSF* simulating the *CPU* is also given.

Example for a locally visible CPU-PSF To simulate the *locally visible CPU-PSF*, which means that the real process is blocked/waiting for an event, can be handled easily by letting it sleep for the desired time.

Implementation For instance in the UNIX environment the "usleep()" function can be used. For details see "unistd.h".

Example for a globally visible CPU-PSF The *globally visible* stubbing of CPU cycles can be realized using "no operation" (NOP) so that

the resource CPU will be used by this *PSF*. They have to be executed in a loop in order to reach the desired time consumption. The main problem is that the "used cycle" should not affect any other part of the system, e.g. the data L1-Cache of the processor. Additionally the NOPs and also the loop have to be protected against compiler optimizations.

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 1µs while looping around the empty statement ";".

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

Listing 1: example implementation of a *globally visible CPU-PSF*

This is only a simplified example and could lead to non-reproduceable results if the throttling functionality of the CPU is switched on.

3.3.2 Memory-PSF

Memory with respect to *PSF* refers to the volatile storage. More in detail the different types of memory described in this sections are:

- main memory
- caches
- registers

Main Memory The delay time of the main memory strongly depends on the architecture [21]. These dependencies can be handled in the setup of the *dynamic performance stubs* using the *calibration functions*. Although the paging/swapping mechanisms are often handled as I/O, it will be integrated to this *PSF* category. The reason for this is the coherence between the lack of memory and paging/swapping. Moreover, our definition about *I/O-PSF* does not match due to the fact that there is no systemcall caused by the application to do paging/swapping. Additionally it is recommended to deactivate paging/swapping in high performance applications to increase the predictability of the systems behavior.

The classification into *locally-* and *globally visible* can also be done. An only *locally visible* interaction of the *PSF* for the main memory can be realized by locking any size of memory which is only locally available to the stub. As long as enough memory remains for the system, no swapping or paging has been caused by the locked parts. The system will "stay stable" and the stub can use it is own memory.

For a *globally visible* influence the same methodology can be used. Instead of protecting the system from paging/swapping this behavior will be forced by locking a large amount of memory. This will lead to memory misses and results in hard disc accesses.

Caches Current processors have cache hierarchies, e.g. the L1-, L2- and L3-Caches. Depending on the model there can also be different L1-Caches for data and instructions [22]. The architecture which is currently used within our research project does not provide a L1 Instruction Cache. Therefore in this first approach only L1 Data Caches will be taken into consideration. The system provides the possibility of controlling the caching of instructions and data through different dedicated registers and several bits. Also the possibility to flush or to disable the caches is given. These properties can be used for the *globally-* and *locally visible* manipulation of the caches. For the *globally visible* manipulation the already mentioned feature of disabling the caching mechanism can be used. This will lead to an in-

creased execution time because all data have to be taken from the main memory. Tagging cache lines as modified and working on the same data will lead to a cache miss although the data does not really need to be changed. This behaviors can be used for only interacting *locally visible*.

The aforementioned methods represent just a short introduction to the manipulation of caches. There are much more different possibilities for modifying the caching behavior, e.g. different caching algorithms, implicit/explicit caching [22] or thrashing the "direct mapped" cache [23].

Processor Registers In [24] the processor registers are divided into "user-visible" and "control and status" registers. Here only the "user-visible" will be taken into account. Additionally there is only a *locally visible* impact. This can be realized by e.g. using variables which are tagged as "volatile". Here the variables will be stored in the memory instead of the registers. This can be interpreted as "switch-off' the register usage and will end up in additional access time to the variable.

Additionally the stack will handled be in this section. Stack related problems can have their origin in large arrays as local variables [25].

Further and more detailed studies in this area of processor registers will be conducted. Also different architectures and methods such as register-slicing have to be evaluated.

3.3.3 I/O-PSF

I/O in computer science describes the way to handle the input and output values. Regarding to performance improvement of user applications I/O can be described as the interface between the application and the kernel realized by specific systemcalls [25, 26]. During the execution of these calls the process is blocked due to "I/O" and will be scheduled out, which means the CPU will be freed from the process until the call has been finished and the process will be scheduled in again [25]. Systemcalls cover all interactions between the kernel and the user application this includes also the memory and the network. However, these parts are already discussed in other sections of the *PSF* and will not be handled here. Therefore only the secondary memory, e.g. hard disks are taken into account. Further and more detailed studies in this area of *I/O-PSF* are necessary.

3.3.4 Network-PSF

The *Network-PSF* will handle possible performance parameters specific to the network, especially the data transferred from and into the system. Also this part of the *PSF* can be divided into *globally-* and *locally visible* implementation.

The first of these two parts can be simulated by e.g. adding and removing network messages from the receiving and sending queues. Another way to manipulate the whole system can be realized with a producer/consumer scenario which can be built using *global dynamic performance stubs*. An example to reduce the network load can modeled as follows. The SUT will only send a fingerprint of this message and the CUS will interpret the message and will answer in a proper way to keep the system running. Therefore additionally to the CUS the SUT has to be changed which will probably add some overhead.

The second (*locally visible*) manipulation using the *Network-PSF* part can be to discard the incoming messages without processing them and using already prefetched data for answering. Additionally there are several other approaches to improve the speed of the network, e.g. switching to an other IP stack, using a different protocol or to change the properties of the sending behavior [25].

3.4 Performance Measurement Functions (PMF)

The *performance measurement functions* are a set of different functions for measuring performance behavior and aimed to a proper calibration of the *PSF* to the target. They can additionally be used for basic evaluations in-

side of the SUT, CUS or *global dynamic performance stub*. The functions will be mainly based on already existing functions such as the "getrusage()" of "sys/resource.h" in Linux. Therefore the *performance measurement functions* can be considered as a wrapper classes to provide a common interface to the user and to hide the internals. As long as it is possible they will be inlined in order to reduce overhead.

3.5 Calibration Functions (CF)

Some different values, e.g. the "TIME" (see also Section 3.3.1) value in listing 1, have to be set up in order to provide the suggested simulation properties. This can be done using the *calibration functions*. They provide an environment for the *PSF* and *performance measurement functions*. The idea behind these functions is to execute the *PSF* with several different input values and to trace the according output using the performance measurements provided by the *performance measurement functions*. The calculation of the desired values will also be done inside of the *calibration functions*. Using the results will provide a proper setup for the usage of different *PSF* in order to stub the CUS. The *calibration functions* are providing additional functionality such as they will report if something unexpected was happening. In this scenario they will also give hints on how to improve the measurement for a proper calibration. As an example: a context switch happened while the configuration of the *global CPU-PSF*. The *calibration function's* trace includes a warning that a context switch was happening. Additionally it will give a hint like raising the priority of the *dynamic performance stubs* in order to get valid results.

4 Using Dynamic Performance Stubs - Steps towards a Methodology

In this section first steps on how to use *dynamic performance stubs* are presented. Due to the fact that performance improvement is a highly specialized task, where it is hard to describe a standard procedure, only a possible method can be given, which can be used in many cases, but not all. However, we extend this method by some special operations, which can be seen as a toolset to create *dynamic performance stubs*.

4.1 Standard Method

As a fundamental prerequisite a reproducible automated test of the system has to be available. The system's use has to be restricted to the task of performance optimization. Without such a test procedure it is not possible to reproduce the performance results and thus to validate the results or even more to isolate performance bottlenecks. In addition the *calibration function* has to be parametrized according the used hardware platform as sketched in Section 3.5.

The method for performance optimizations using dynamic performance stubs can be described in the following steps:

1. Identification of performance bottlenecks.

   As a first step some possible bottlenecks of the origin software have to be identified. First a performance measurement of the whole system has to be realized. The delivered performance data has to be interpreted and some possible bottlenecks have to be identified. This can be achieved, if all performance data are available in a very detailed level. So a "drill-down" method on the delivered performance data can be used. Please note that the success of this step strongly depends on the experience and the in depth knowledge of the software of the analyzing person. As a result of this step a list with possible performance bottlenecks has to be generated. For further information it is refered to [5, 8].

2. A cost-benefit analysis of optimizations should be done.

   Please note that in a first step this ratio only might be estimated. The result of

this step is a prioritized list of bottleneck candidates.

3. Now a *dynamic performance stub* has to be generated.

   This can be done with the following steps:

   (a) The CUS has to be determined. This also includes a necessary level of abstraction, especially which functions have to be stubbed.

   (b) The functional behavior of the CUS has to be simulated. This will be done by the realization of the *simulated software functionality*.

   (c) The performance behavior of the CUS has to be determined and simulated. The simulation will be realized by the *PSF*.

   (d) The correct function of the *simulated software functionality* and the *PSF* have to be validated. So performance measurements have to be done with the stubbed functionality. The results should be in the same range as the original performance results. If this is not the case, the *PSF* has to be modified. This might also happen if the analysis for this bottleneck candidate is not sound and as a consequence it has to be reconsidered.

   (e) As an additional check a performance measurement with only a *simulated software functionality* can be done and the resulting data should be analyzed very carefully. In addition the next bottleneck might be visible.

4. Several measurements changing the performance behavior of the stub should be realized.

   The performance data has to be analyzed carefully. This can be done as described in Step 1. Additionally several different charts can be drawn. They will probably indicate different system behaviors such as a changeover from CPU to memory bound.

   A cost-benefit analysis using the possible gain can be applied and the effort for improving the performance bottleneck can be estimated based on the evaluation data. As a further result there might appear new bottlenecks, which should be checked using the method in Step 2. If no new bottlenecks can be found, the next step can be taken.

5. Based on this data the candidates for spending optimization efforts can be chosen and the work on improvement can be started.

6. The optimized components should be included in the software and a new performance measurement should be started.

   The achieved data should be compared with simulated data. If there is a huge discrepancy the method for finding bottlenecks or even *PSF* should be corrected.

7. If the software has still not the desired performance, goto Step 1.

4.2 Extension of the Standard Method

In many cases the aforementioned method can be used. However, it can be seen that performance analysis is a highly sophisticated and specialized task. In the following section some possible techniques or alternatives to the method described before have been sketched. These can be seen as some ideas which might help on the overall task like a partially filled toolbox.

4.2.1 Mixture of PSF

Sometimes an isolated performance behavior is not sufficient for simulation. E.g. using only the *CPU-PSF* might lead to false results, if the component uses in addition to a possible heavy usage of the CPU also disk I/O very often. This is the case especially in bigger CUS. A solution is to combine several different types

of *PSF* in order to yield a more accurate simulation result. However, the degree of combination might be analyzed very careful. This is especially on bigger CUS no easy task.

4.2.2 Full and Partial Stubs

Another problem appears on bigger CUS, too. The components behavior can not be simulated in a realistic way by a simple stub, where a collection of input values delivers an output value. Usually it might be an arbitrary interleaved mixture of calling subroutines, internal calculations and delivering results. This can be realized by extending the stub with additional *simulated software functionality* which is calling the real procedures and discarding the results from it or storing them for further analysis, e.g. in a hash table.

The same technique can be used for functions which could not be optimized and therefore is no need to be simulated.

4.2.3 Idealized Measurements

The stubs can also be used to simulate almost ideal performance behavior of the CUS. This is useful for a feasibility study or for verifying the specified performance targets of third party software, e.g. middleware and can be realized by only using the capabilities of the *simulated software functionality* without inserting other functions as *PSF* or *performance measurement functions*. This will tear down the cycles to a minimum and therefore the "other" software modules can use almost every resources in total. Of course, the operating system will also contribute it is cost to the total load.

4.2.4 Load and Stress Tests

Another possibility is the opposite to the idealized measurements. Here the system will be stressed by adding additional performance bottlenecks to evaluate the system behavior under a high load. It can be realized by using a *global dynamic performance stub*, e.g. a stub which is using the *globally visible* functionality of the *CPU-PSF* combined with a high priority. This will increase the CPU usage and therefore will stress the system. Depending on the software, overload routines will be performed in such situations. So the behavior of the application on the borderline can be examined and new bottlenecks can be identified.

4.2.5 Hidden Bottlenecks Detection by Zero Bound CUS

Hidden bottlenecks are bottlenecks which are existing but can only be seen after removing the current one. They are also reducing the throughput but it does not count to the overall performance due to the stronger influence of the first bottleneck. Specialists often "over optimize" the software module which leads to that the hidden bottleneck becomes critical [1].

The following additional techniques help in detecting hidden bottlenecks. The CUS will be optimized in a non realistic way by setting the performance values of the *PSF* to zero. So a new performance measurement will help to make the next bottlenecks visible.

4.2.6 System Bounds

If all local optimizations do not lead to the required results or are too expensive, sometimes a global optimization might be chosen, for instance using a faster CPU or a bigger memory. To check whether global restrictions are there *dynamic performance stubs* can help. E.g. after evaluating several runs with different performance parameters a chart can be drawn. This can point to a system restriction, e.g. the system starts swapping/paging due to a lack of memory.

4.2.7 Global vs. Local Stubs

As can be seen in Figure 1 there are two different occurrences of *dynamic performance stubs*.

First they can work *locally* which means to stub a software unit. This procedure is handled along the paper and will not be described more in detail in this section.

The second way of using the *dynamic performance stubs* is *globally*. Therefore a software

application (module) has to be written which will affect the whole system in a desired way e.g. by generating CPU load.

As an example: a module will be created which compares the current usage of the CPU with a default value. If the current value is less than the default, the module will use the CPU, e.g. with the *CPU-PSF* and tries to adjust the usage by generating load. Otherwise it will not do anything. As a result global considerations can be realized. It can be checked whether the throughput of the whole system is still on a limit. E.g. if after a linear increase of the CPU load by a global stub lead to a non-linear increase of the runtime of the process to be examined. This might be an indication that the system throughput is too low to handle all requests. Then an increase of the CPU power might help. However, this might not be an option in all projects, but also then this examination result helps in gaining a better understanding of the systems behavior and in consequence of the possible performance problem.

Another possibility for the usage of a *global dynamic performance stub* is to increase the number of context switches. For this a software module can be created which only wakes up and goes back to sleep again. If it will only sleep for a short time and is executed with a high priority the system will do a context switch. This module can be realized using the *CPU-PSF*.

5 Advantages

As mentioned the *dynamic performance stub* can be used for a cost-benefit analysis. This will also lead to a balance between optimization effort and the achievable gain in the system. Of course a higher optimization of the software module will also lead to a higher performance within the complete system but the effort for the additional gain in this case might be to big. This approach can lead to a more gain-oriented optimization. This point will end up in better maintainable and structured code. General drawbacks of performance improvements such as a poor maintainability and badly structured code is described in [18, 27]. Which can be reduced by the presented approach.

Additionally, knowing the optimization effort can lead to results earlier, because some possible improvements does not have to be done due to system dependencies [1].

As described in Section 4.2.4 “load & stress” tests are possible. This can also be used for testing functions, e.g. overload routines under “real” conditions because the CPU utilization will be raised up by a *global dynamic performance stub* module. As mentioned in Section 4.2.3 the similar methodology can also be used for idealized measurements regarding the performance behavior of the CUS.

The *dynamic performance stub* can be used to identify hidden bottlenecks as described in 4.2.5.

6 Restrictions

The *dynamic performance stub* has also some drawbacks which will be described in this section. The first to mention is the danger of wrong results of the initial performance measurements. If this error happens all further results of the measurements using the stub can also be wrong. This can lead to a lot of time wasted in measuring, building and evaluating the performance. As mentioned in [5] all results should be handled with care until they are validated.

Also with proper initial measurements and a proper stub setup the results can be misleading, e.g. because of the introduced overhead. Therefore as already stated all results should be handled with care.

Creating a *dynamic performance stub* means effort which can require additional costs. The gain of this method strongly depends on the system, it is performance behavior and the effort to be spent for optimizations.

Large projects often take a lot of additional effort for performance stubbing, each iteration of stubbing one element requires a change of parameters and the repetition of the build- and

performance measurement process as well as the evaluation.

Within the software life cycle the interfaces and the messages of the software can change. This means probably that a stub has to be adapted after each of this changes.

There is a lot of additional effort as result of measurement operations. However, using the described method should decrease the overall improvement effort especially on large software systems.

7 Conclusion and Future Work

In this paper a method for supporting performance improvement operations has been shown. Therefore the concept of *dynamic performance stubs* has been introduced, which is a mixture of the stubbing mechanism and simulation of performance behavior.

Using *dynamic performance stubs* following advantages for performance improvements can be achieved:

- The results of possible performance optimizations can be estimated with an increased confidence level, because a complete program execution with a simulated performance optimization can be achieved before the optimization has been realized. As a consequence a validated estimation of a performance improvement can be given before the effort of a concrete optimization has been spent. This optimization effort can sometimes be several man months instead of the effort for generating a performance stub which in most cases can be measured in hours. Since the benefit of performance improvement can now be determined, a valid cost-benefit analysis for improvement operations can be calculated, if the effort for each operation has been estimated. So different candidates for performance improvements can be prioritized and such the effort of improvement can be spent on a section with a big return on investment.

- Using *dynamic performance stubs* also the necessary performance gain can be determined. Sometimes an improvement of e.g. the reduction of the execution time of a bottleneck to 50% does lead to the same execution time for the whole application as a reduction to 75%, because other components will be new bottlenecks. So by *dynamic performance stubs* the necessary ratio of improvement operations can be determined and in consequence an overengineered performance improvement can be avoided.

- Also the “hill climbing phenomena” can be avoided. Usually, on mountains hiking only the next peak can be seen in advance and later higher hilltops on the way are hidden. This usually happens with performance bottlenecks too. If you have reduced one, the next bottleneck appears [1]. Here *dynamic performance stubs* can be used to see the upcoming bottlenecks in advance.

In this paper the basic concepts behind *dynamic performance stubs* and a method using performance stubs in the area of performance improvements has been sketched. However, the concept and methodology has to elaborated in the following points:

- The calibration functions have to be implemented to reduce hardware dependency within one platform. Especially some hardware related topics e.g. CPU throttling has to be elaborated.

- The remaining *PSF*s have to be implemented, especially the *I/O-PSF* has to be elaborated. Also several additional simulation parameters have to studied and simulated, e.g. data/instruction TLBs, L1 Instruction Caches, branch predictions or paging/swapping as well as pipelining effects.

- *PSF*s have to be implemented for different platforms to realize hardware indepen-

dence. Also multiprocessor architectures have to be taken into account.

- The combination of different *PSF*s has to be examined to simulate realistic system behavior.
- The methodology of the usage of performance stubs has to be elaborated more in detail and to be verified by different case studies. Especially a method how to realize decomposition of performance targets to smaller components has to be created.
- A methodology for stubbing non-deterministic functions has to be elaborated as well as the evaluation techniques, e.g. statistical evaluations of the performance measurement results.
- A wider range of evaluation techniques and the resulting conclusions, e.g. identifying changeovers from system bounds should be studied.
- The loss of precision for performance measurements using *PSF* compared with realistic optimizations has to be examined.
- The possibility to support the software development process by using *dynamic performance stubs* has to be studied.
- All functions have to be supported by a toolset, which has to be implemented.

All described open points will be examined in the ongoing research project.

8 Acknowledgment

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

For careful reading and providing valuable comments on draft versions of this paper we would like to thank Prof. Dr. Stefan Hahndel, Marinus Luegmair, Dr. Alexander Ost and Fabian Wohlschläger.

References

[1] S. McConnell. *Code complete: A practical handbook of software construction*. Microsoft Press, Redmond, WA, USA, 2 edition, 2004.

[2] G. Marin and J. Mellor-Crummey. Application Insight Through Performance Modeling. In *26th IEEE International Performance Computing and Communications Conference (IPCCC'07)*, New Orleans, April 2007.

[3] D. J. Reifer. The smart stub as a software management tool. *SIGSOFT Softw. Eng. Notes*, 1(2):4–8, 1976.

[4] Satish Chandra Gupta. Need for speed – eliminating performance bottlenecks. IBM (online): `http://www-128.ibm.com/developerworks/rational/library/05/1004_gupta/index.html`, 2007. [April 25, 2007].

[5] R. Jain. *The art of computer systems performance analysis*. Wiley and sons, Inc., 1991.

[6] J. J. Marciniak. *Encyclopedia of Software Engineering*. John Wiley & Sons Inc, 2 edition, 2002.

[7] P. J. Fortier and H. E. Michel. *Computer Systems Performance Evaluation and Prediction*, volume 1. Digital Press, Burlington, 2003.

[8] D. J. Lilja. *Measuring Computer Performance: a practioner's guide*. Cambridge University Press, New York, 2000.

[9] A. Schmietendorf and A. Scholz. *Aspects of Performance Engineering - An Overview*, pages IX – XII. Springer-Verlag Berlin, Heidelberg, 2001.

[10] 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.

[11] P. Liggesmeyer. *Software-Qualität : Testen, Analysieren und Verifizieren von Software*. Spektrum Akademischer Verlag GmbH, Berlin, 2002.

[12] I. Sommerville. *Software Engineering*. Addison-Wesley, 6 edition, 2001. german redaction.

[13] N. H. Gunther. *The practical performance analyst*. McGraw-Hill Education, 1998.

[14] C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, 2 edition, 2003.

[15] GNU gprof - The GNU Profiler. online: `http://gnu.org/software/binutils/manual/gprof-2.9.1/html_mono/gprof.html`, 2007. [April 16, 2007].

[16] Linux Trace Toolkit. online: `http://www.opersys.com/LTT`, 2007. [April 16, 2007].

[17] OProfile - A System Profiler for Linux. online: `http://oprofile.sourceforge.net`, 2007. [April 16, 2007].

[18] F. Agner. 1. Optimizing software in C++: An optimization guide for Windows, Linux and Mac platforms. online: `http://www.agner.org/optimize`, 2006. [last update: July 5, 2006].

[19] Optimize Options - Using the GNU Compiler Collection (GCC). online: `http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`, 2007. [April 20, 2007].

[20] J. Hughes. Performance Engineering throughout the System Life Cycle. Technical report, SES Inc., 1998. [April 04, 2007].

[21] R. Srinivasan and O. Lubeck. MonteSim: A Monte Carlo Performance Model for In-order Microarchitectures. *ACM SIGARCH Computer Architectur News*, 33(5):75–80, December 2005.

[22] Intel Corporation. *IA-32 Intel® Architecture Software Developer´s Manual - System Programming Guide, Part I*, June 2006.

[23] C. Siemers, M. Eckert, and A. Lauchner. Prozessor-Technologie. *tecchannel compact*, (3/04), 2004.

[24] W. Stallings. *Computer Organization & Architecture: Designing for Performance*. Prentice-Hall, 6 edition, 2003.

[25] M. E. Lee. Optimization of Computer Programs in C. online: `http://www.prism.uvsq.fr/~cedb/local_copies/lee.html`, 1999. April 25, 2007.

[26] Robert Love. *Linux-Kernel-Handbuch*. Addison-Wesley Verlag, 2005.

[27] J. M. Newcomer. Optimization: Your Worst Enemy. online: `http://www.codeproject.com/tips/optimizationenemy.asp?print=true`, 13 August 2000. [May 02, 2007].

Biograghy

Dipl.-Ing. (FH) Peter Trapp is a research assistant at the Institut für Angewandte Forschung of the Fachhochschule Ingolstadt. His research interests are in the area of performance measurements and analysis and in computer security.
Peter Trapp was born in 1978 at Neuendettelsau, Germany. After his graduate in Eletrical Engineering and Information Technology at the Fachhochschule Ingolstadt he was employed at BMW in the bluetooth handsfree set development department.

Dr. Christian Facchi is Professor for SW Engineering, Distributed Systems and Mathematics at the Fachhochschule Ingolstadt, Germany since 2004. His major research interests are in the area of performance analysis and modeling, embedded systems, testing of Software and formal description techniques.
Dr. Facchi was born in 1964 at Munich, Germany. He holds a doctoral degree and a diploma degree from the Technische Universität München, Germany both for Computer Science. Before he changed to the Fachhochschule Ingolstadt he has been employed at Siemens in the Mobile Phones development department for 9 years. During his employment at Siemens Mobile Phones he was the head of worldwide strategy for SW development environments.

Impressum

Herausgeber
Der Präsident der
Fachhochschule Ingolstadt

Esplanade 10
85049 Ingolstadt
Telefon: 08 41 / 93 48 - 0
Fax: 08 41 / 93 48 - 200
E-Mail: info@fh-ingolstadt.de

Druck
Hausdruck

Die Beiträge aus der FH-Reihe
"Arbeitsberichte/ Working Papers"
erscheinen in unregelmäßigen Abständen.

Internet
Dieses Thema können Sie, ebenso wie die früheren Veröffentlichungen aus der FH-Reihe "Arbeitsberichte - Working Papers", unter der Adresse *www.fh-ingolstadt.de* nachlesen.

ISSN 1612-6483