Using CPU Stubs to Optimize Parallel Processing Tasks: An Application of Dynamic Performance Stubs

Peter Trapp, Markus Meyer, and Christian Facchi
University of Applied Sciences
Ingolstadt
Ingolstadt, Germany
{peter.trapp, markus.meyer, christian.facchi}@haw-ingolstadt.de

Abstract—Dynamic performance stubs provide a framework for the simulation of the performance behavior of software modules and functions. Hence, they can be used as an extension to software performance engineering methodologies. The methodology of dynamic performance stubs can be used for a gain oriented performance improvement. It is also possible to identify “hidden” bottlenecks and to prioritize optimization possibilities. Nowadays, the processing power of CPUs is mainly increasing by adding more cores to the architecture. To have benefits from this, new software is mostly designed for parallel processing, especially, in large software projects. As software performance optimizations can be difficult in these environments, new methodologies have to be defined. This paper extends and improves the methodology of CPU stubs and applies it to multi-core environments and parallel processing. The method is evaluated by means of a proof of concept. We were able to show that CPU stubs can be used to identify performance bottlenecks in parallel processing environments and to quantify the gain of different performance improvements. Hence, a new methodology for gain oriented optimization of CPU bound parallel processes has been validated in a casestudy.

Keywords—software performance optimization; CPU bound applications; multi-core; parallel processing; dynamic performance stubs

I. INTRODUCTION

Dynamic performance stubs have been introduced in [1]. They can be used for “hidden bottleneck” detection, and, by demonstrating the level of optimization potential, a cost-benefit analysis can be performed as well. Dynamic performance stubs extend the methods known from software performance engineering by simulating various levels of system load. This leads to more gain-oriented performance optimizations. The dynamic performance stubs can be used within the software development cycle of large software systems, e.g., in telecommunication systems.

Nowadays, many system architectures achieves higher throughput by using multiple cores. Hence, the application has to be able to do parallel processing to fully utilize the available capacity of the system. As multi-threaded and parallel processes are difficult to optimize, new methodologies in the area of software performance engineering have to be defined. The methodology of dynamic performance stubs can be used to optimize CPU bound applications by using CPU stubs.

A. Dynamic Performance Stubs

The idea behind dynamic performance stubs is a combination of performance improvements [2]–[5] in already existing modules or functions and the stubbing mechanism from software testing [6], [7]. The performance behavior of the component under study (CUS) will be determined and replaced by a dynamic performance stub. This stub can be used to simulate different performance behaviors, which can be parameterized. The optimization expert can use these to analyze the performance of the system under test (SUT). This procedure relates to stubbing a single software unit. Hence, it will be called “local”. Therefore, a “local stub” has to be built. The dynamic performance stub can also be used to change the behavior of the complete system. A software module has to be created, which interacts “globally” in the sense of influencing the whole system instead of a single software component. This stub will be called a “global stub”.

Figure 1. Interactions of “Dynamic Performance Stubs”

Figure 1 sketches the design and the interaction between a real system on the left and the dynamic performance stubs on the right side. The unfilled arrowhead indicates 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 replace a software unit.
The framework of the dynamic performance stub consists of the following parts, which is 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 ability to simulate the performance behavior of the replaced CUS. They are divided into four categories:
  - CPU
  - Memory
  - I/O
  - Network

As an example, the CPU PSF can be used to simulate the processing behavior of the application and hence, the CPU utilization of the process. Please refer to [1] for more information.

- 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 already provided by the system.

- calibration Functions (CF)
  In order to provide trustworthy results, the stubs have to be adjusted to a dedicated system. This can be done using the calibration functions.

For more detailed information on dynamic performance stubs, the reader is referred to [1].

CPU Stubs: CPU stubs, as a special subset of dynamic performance stubs, can be used to handle CPU bound systems. Therefore, a general approach to parameterize the runtime behavior and CPU usage has been achieved as well as a possible realization has been defined in [8]. Additionally, a methodology for evaluating and creating these stubs is provided in [9].

Memory Stubs: Memory stubs are a special subset of dynamic performance stubs regarding the memory usage. They can be used to simulate the memory performance behavior of a suspected software bottleneck. Hence, they combine the simulated software functionality and the memory performance simulation functions [10].

B. Content of the Paper

This paper describes the application of CPU stubs to multi-core architectures to optimize CPU bound parallel processing tasks. Thus, the methodology, as provided in [9], is shortly discussed and significantly extended (Section III). Afterwards the concept of CPU stubs is used to study and optimize parallel processing tasks (Section IV) by means of a proof of concept. The results of the study as well as a summary is described in Section V.

II. CPU Stubs

CPU stubs have been introduced in [1] and extended in [8], [9]. They can be used to simulate the performance behavior of CPU bound applications. Thus, they consist of two simulation elements: system influencing- and system non-influencing CPU PSF.

- System influencing
  This functionality simulates the process execution while the process is running. Regarding the process states [11], this can be seen as “Running”.

- System non-influencing
  This functionality simulates the behavior of a process that has been delayed for any reason, e.g., because of a scheduling event or a waiting for I/O. This functionality can be seen as “Blocked” or “Ready” regarding the process states [11].

More details about the simulated CPU states can be found in [9]. An example implementation of these functionalities as well as a description on how to setup and use CPU stubs can be found in [8], [9]. Please note that, this methodology differs significantly from [9] because of its adaption to parallel processing systems.

III. Methodology

This section describes the methodology for using CPU stubs in parallel systems.

1. Determination of the CPU bottleneck: The SUT has to be defined and a suspected bottleneck (CUS) has to be identified, which is done by common software performance engineering (SPE) [12], [13], e.g., profiling or tracing. Now, several performance indicators have to be determined:

- \( t_{\text{CUS}} \): Time spent in the bottleneck (CUS).
- \( t_{\text{SUT}} \): Time spent in the software module or function (SUT) from which the CUS is part of.
- \( t_{\text{cy}}_{\text{busg}} \): Time spent in the CUS using the CPU. It includes the user-mode time as well as the system-mode time, see also [11].
- \( t_{\text{CUS waiting}} \): Time spent in the CUS waiting to be scheduled, see also: process state “Ready” in [11].
- \( t_{\text{CUS blocked}} \): Time spent in the CUS waiting for an event, see also: process state “Blocked” in [11].

The measured values have to be deterministic within several performance test runs.

2. Validate CPU Bottleneck: In this step, a simple validation of the chosen CUS will be done. Thus, the system influencing CPU PSF is inserted in front of the CUS and the performance measurements will be repeated increasing the time spent in the PSF (\( t_{\text{PSF}} \)). The measured time of the SUT mainly follows one of the diagrams given in Figure 2.

In Figure 2a the increase of the system influencing CPU PSF leads to an arithmetically increasing amount of time spent in the SUT. Therefore, the CUS seems to be a CPU bottleneck. Hence, the next step can processed.
In the other case, Figure 2b shows that an increase in the execution time of the CUS does not increase the time spent in the SUT for \( t_{PSF} < t_{limit} \). This points out that the CUS is no bottleneck for the system. Another CPU bottleneck has to be searched (Step 1).

This step can be done to remove overhead as it excludes the CUS from being mistaken as a CPU bottleneck easily. This step is optional.

3. Study the Bottleneck Performance Behavior: The value \( t_{CUS}^{\text{blocked}} \), which has been determined in Step 1, will be used to evaluate the CPU utilization of the CUS.

A value of \( t_{CUS}^{\text{blocked}} = 0 \) means that there are no waiting periods triggered by the CUS while executing. So, the process will not be interrupted by the CPU except there are external events, e.g., scheduling. In this case, the methodology can be used as provided.

A value of \( t_{CUS}^{\text{blocked}} > 0 \) means that the process switches to the “Blocked” state. Here, the trace files recorded in Step 1 have to be studied further, in order to identify successive working and waiting periods of the process. The following steps of this methodology have to be done for every working period starting from the biggest to the smallest working period. The waiting period will be simulated with the system non-influencing CPU PSF. The simulated waiting time will normally be constant, if no further reduction caused by optimizations of this time period can be expected.

4. Flat CPU Stub - Evaluate the Optimization Potential: Now, a flat CPU stub will be used to determine the optimization potential. A flat CPU stub is a dynamic performance stub, which only simulates the functional behavior of the CUS using the simulated software functionality. Hence, it only introduces small overhead in the system and can be used to simulate the ideal time behavior of the CUS. This can be used to analyze the maximum performance gain of the SUT as it is not the same as \( t_{CUS} = 0 \), especially in multi-core or parallel processing environments. As the final result often depends on several in parallel working threads or processes. Therefore, the following values have to be measured:

- \( t_{\text{SUT}}^{\text{stub}} \): Time spent in the flat CPU stub.
- \( t_{\text{SUT}} \): Time spent in the SUT including the flat CPU stub.

An indicator of the possible optimization amount can be evaluated by calculating \( t_{\text{cus}}^{\text{reduced}} = t_{\text{CUS}} - t_{\text{flat}} \) and \( t_{\text{SUT}}^{\text{iter}} = t_{\text{SUT}} - t_{\text{flat}} \). This is the time, which has been reduced in the CUS. The \( t_{\text{reduced}} \) value describes the total possible optimization gain. If the CUS is executed more than once in sequence, the maximum number of iterations per CPU (iter) has to be evaluated. Hence, \( t_{\text{cus}}^{\text{reduced}} \) * iter and \( t_{\text{SUT}}^{\text{reduced}} \) has to be compared. The values \( t_{\text{CUS}}^{\text{reduced}} \) and \( t_{\text{SUT}}^{\text{reduced}} \) are taken from Step 1. Now, the calculated values can be compared and the following cases can be evaluated:

- \( t_{\text{CUS}}^{\text{reduced}} = t_{\text{SUT}}^{\text{reduced}} \): This means that the SUT directly depends on the CUS. Hence, there are no system dependencies, i.e., “hidden bottlenecks”. Additionally, no “over optimization”, as described in [14], can be done. The more time optimized in the CUS the better it is. In this case, the next step of this methodology is Step 7, i.e., optimize as much as possible. However, in case of an expected hardware bottleneck, Step 5 can be done. This behavior is typically for batch or procedural processing in single core environments.
- \( t_{\text{CUS}}^{\text{reduced}} < t_{\text{SUT}}^{\text{reduced}} \): In this case, the possible optimization amount is less than the time spent in the CUS. Thus, there are system dependencies, which have to be studied further. The next step of this methodology can be done. This behavior can mainly be seen in multi-core and parallel processing systems. As there might be parallel threads or processes, which additionally delays the execution after the actual bottleneck has been reduced.

As it is only an indicator, the time \( t_{\text{SUT}}^{\text{reduced}} \) delivers no information about the amount of optimization, which has to be done in the CUS, especially for \( t_{\text{CUS}}^{\text{reduced}} > t_{\text{SUT}}^{\text{reduced}} \).

5. Idle CPU Stub - Evaluate System Dependencies: Here, the flat CPU stub will be extended using the system non-influencing CPU PSF. This is called an idle CPU stub. The total simulated time is the total processing time of the
CUS ($t_{CUS}$). Hence, the following equation holds $t_{CUS} = t_{SUT}^{busy}$. Where, $t_{SUT}^{busy}$ is the time spent in the idle CPU stub. Now, the performance measurements will be redone and the $t_{SUT}^{idle}$ value, which is the total execution time of the SUT including the idle CPU stub, shall be recorded.

 Dependencies between an idle CPU stub and the system can be evaluated using the values: $t_{SUT}^{idle}$ and $t_{SUT}$. Thus, the total execution time of the original SUT will be compared to the execution time of the SUT using the idle CPU stub. The following cases can be separated:

- $t_{SUT}^{idle} = t_{SUT}$;  
  This means that the total execution time of the SUT has not changed due to the usage of the idle CPU stub. Whereas, the idle CPU stub only uses the CPU at the very first beginning and then hands the CPU over to the system. However, the total execution time of the SUT has not been changed. Hence, the conclusion that no other process is blocked by the CPU can be done. Therefore, adding CPUs to the system does not provide a significant performance improvement. Nevertheless, as of Step 2, the CUS is the bottleneck.

- $t_{SUT}^{idle} < t_{SUT}$;  
  Here, the total execution time of the SUT decreases by using an idle CPU stub. Therefore, further processes are at least partially available in the “Ready” queue. In this case, these processes can be executed earlier. Therefore, the total execution time decreases. An optimization of the CUS as well as an additional CPU decreases the total execution time.

This step evaluates dependencies between running processes in the system and the CUS. Moreover, information about the influence of adding CPUs to the system can be achieved. However, the measurements do not provide any information whether a faster CPU will increase the total execution time. Albeit expected that a faster CPU will increase the total execution time as the process is CPU bound.

6. Busy CPU Stub - Cost Estimation: The flat CPU stub will be extended with the system influencing CPU PSF. Now, the performance measurements will be repeated and the time spent in the system influencing CPU PSF ($t_{PSF}$) will be varied from zero to the total execution time of the CUS ($t_{CUS}^{busy}$). The following values have to be measured:

- $t_{SUT}^{busy}$: Time spent in the busy CPU stub.
- $t_{busy}$: Time spent in the SUT including the busy CPU stub.

Using these results, two different types of bottlenecks can be distinguished:

- Total Bottleneck:  
  In this case, the measured values of the execution time from the SUT is linearly increasing. Thus, an optimization of the CUS will always result in an improvement of the execution speed and, therefore, decrease the latency. This result should have been already achieved in Step 4.

- Limited Bottleneck:  
  If the processing of the SUT depends on other functions respectively on their results, the graph might look similarly as given in Figure 3. The graph is split in two parts. In the first part, $t_{PSF} \leq t_{limit}$, the time of the SUT is constant at a minimum value ($t_{limit}^{min}$). Within this area, the chosen CUS is no bottleneck to the system as an increasing in the amount of processing ($t_{PSF}$) does not lead to an increased execution time ($t_{SUT}$). At $t_{limit}$, the behavior of the CUS changes to a CPU bottleneck. As can be seen in the figure, the time spent in the SUT increases along the time spent in the system influencing CPU PSF ($t_{PSF}$). This evaluation shows that an optimization of the bottleneck can only decrease the latency in the SUT to a given value ($t_{SUT}^{min}$).

These information can be used to identify “hidden” bottlenecks, e.g., a “hidden” bottleneck appears at $t_{limit}$ of Figure 3. This limit is basically the maximum, which can be achieved by an optimization of the CUS. Hence, it can be compared to a changeover in the critical path (see also [15]). Additionally, the information can be used for a cost-benefit analysis. Thus, a gain-oriented improvement can be done.

7. Optimization of the Software: Now, the software module or function has to be optimized. Hence, the results from the cost-benefit analysis can be used for a software improvement related to the optimum between cost and effort. Finally, the performance of the software component has to be measured again. A new bottleneck has to be identified (first step) if the results show that the performance targets are not achieved yet.

Summary: The methodology of CPU stubs of [9] has been extended and applied to multi-core and parallel processing environments within this section.

IV. PROOF OF CONCEPT

This section applies the methodology from Section III to a parallel processing application by means of a proof of concept.
A. Test Environment

The application used for the proof of concept is split into three highly CPU bound processing parts. The middle part calculates the results in two different processes and the last part needs both results to complete the calculation. The application runs on a Linux operation system (Kernel 2.6.30.9). The evaluation is done using the Linux Trace Toolkit next generation (LTTng) [16]. For timing measurements, the time stamp counter (TSC) [17] is used. The hardware is based on an Intel Centrino Core 2 Duo CPU. The calibration of the CPU PSF is realized as described in [8].

B. Realization

The described methodology has been applied to the before mentioned application on a step-by-step analysis. The measurements of each step have been done several times in order to evaluate some statistical behavior.

1. Determination of the CPU bottleneck: The SUT is defined as the whole application. The identified bottleneck is supposed to be in the parallel processing part of the application. The measurements have been repeated five times. Table I lists the average value for each parameter as well as the squared coefficient of variation (sqd coeff of var) [18]. The \( t_{\text{waiting}} \) has been calculated. Hence, the squared coefficient of variation is zero.

In this step, the SUT and the CUS have been determined and some more detailed analysis has been done. For a definition of the values see Section III Step 1.

2. Validate CPU Bottleneck: Here, the system influencing CPU PSF has been added to the CUS and the timing behavior of the SUT has been studied. The evaluation of the result has shown a similar behavior as presented in Figure 2a. Hence, the bottleneck has been validated.

3. Study the Bottleneck Performance Behavior: The evaluation of \( t_{\text{CUS \ blocked}} \) has shown that the application is 100% CPU bound, i.e., the process did not stall. Hence, no special steps are necessary in the following steps.

4. Flat CPU Stub - Evaluate the Optimization Potential: In this step, the simulated software functionality has to be build without any PSF. Afterwards, the time spent in the flat CPU stub (\( t_{\text{SUT \ flat}} \)) has to be measured as well as the time spent in the system under test (\( t_{\text{flat}} \)). This measurement has been done five times in order to get some statistical distribution validation.

<table>
<thead>
<tr>
<th>Seconds</th>
<th>Sqd Coeff of Var</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUT</td>
<td>7.21</td>
</tr>
<tr>
<td>CUS</td>
<td>2.48</td>
</tr>
<tr>
<td>CUS _busy</td>
<td>2.42</td>
</tr>
<tr>
<td>t_{\text{waiting}}</td>
<td>0.06</td>
</tr>
<tr>
<td>t_{\text{CUS \ blocked}}</td>
<td>0</td>
</tr>
</tbody>
</table>

The results have been summarized in Table II. As can be seen, the time spent in the stub (\( t_{\text{SUT \ flat}} \)) is approximately zero. The total execution time of the SUT (\( t_{\text{SUT}} \)) is about 0.93 seconds, which has been defined as \( t_{\text{SUT \ reduced}} \), less than the SUT time measured in Step 1 (see Line 2 of Table I). However, the time spent in the CUS (\( t_{\text{CUS \ reduced}} \)) has been reduced by about 2.48 seconds (see \( t_{\text{CUS \ busy}} \) in Line 3 of Table I). As \( t_{\text{SUT \ reduced}} > t_{\text{CUS \ reduced}} \), this is an indicator that there are further influences of the system.

5. Idle CPU Stub - Evaluate System Dependencies: Here, the flat CPU stub is extended by the system non-influencing CPU PSF. This step basically evaluates whether an additional CPU might improve the total execution time. The measured average time of \( t_{\text{idle \ SUT}} \) is 7.15. The comparison of this value with \( t_{\text{SUT}} \) from Step 1 pointed out that an additional CPU would not result in a significantly improved execution time of the SUT. This result can be easily explained, as there are two CPUs for executing two parallel processes and the rest of the system is idle.

6. Busy CPU Stub - Cost Estimation: This step determines the amount of CPU time spent in the CUS, which should be reduced for an ideal optimization. Hence, the system influencing CPU PSF will be successively increased starting from zero to \( t_{\text{CUS \ busy}} \). The values \( t_{\text{STUB \ busy}} \) and \( t_{\text{SUT \ busy}} \) have been measured.

Figure 4 evaluates the \( t_{\text{SUT \ busy}} \) and \( t_{\text{STUB \ busy}} \) depending on the ratio \( \frac{t_{\text{busy}}}{t_{\text{busy}} \_\_\text{ideal \ optimization}} \). Whereas the value for \( t_{\text{SUT \ busy}} \) is increasing arithmetically, the value for \( t_{\text{STUB \ busy}} \) remains constant until 0.6 and then starts to increase arithmetically. Here, a changeover of the bottleneck can be seen. In this case, the CUS is only a bottleneck above 0.6 for the system. Hence, in order to ideally optimize, the time spent in the CUS has only be
reduced to ideal_optimization times \( t_{\text{CUS}} \). Now, a cost-benefit analysis can be done by an effort estimation of the optimization.

7. Optimization of the Software: In this step, we highly “over-optimized” the CUS to clearly identify the “hidden” bottleneck. We were able to reduce the time spent in the CUS to 0.280 micro seconds. However, the time spent in the SUT was still 6.29 seconds, which is close to the expected value as achieved in Step 4. Now, the next bottleneck should be optimized if necessary.

   Summary: The methodology of Section III can be used to optimize parallel processing system. We have shown that, a possible optimization gain can be evaluated and “hidden” bottlenecks can easily be identified. This leads to a more gain-oriented optimization of the software.

V. CONCLUSION AND FUTURE WORK

CPU stubs, which are an element of the dynamic performance framework, enhances the software performance engineering methodologies. In this paper, the methodology for using CPU stubs is significantly extended and applied to multi-core and parallel processing of software systems. Additionally, it is validated by means of a proof of concept. The results show that, CPU stubs can be used to identify performance bottlenecks and quantify a possible optimization gain. Thus, they can be used for a gain-oriented performance optimization. Hence, a new methodology to identify and analyze CPU bottlenecks in multi-core and parallel processing systems has been achieved.

This work can be extended by a real world example in multi-core or parallel processing environments. Additionally, a proof of concept, or even better a real world example, will be used to validate the methodology for bottlenecks with a CPU behavior of \( t_{\text{CUS}}^\text{blocked} > 0 \).

ACKNOWLEDGMENT

This research is granted by Nokia Siemens Networks (NSN). The authors would like to thank the LTE group, especially Oliver Korpilla, Karl Mattern, Helmut Voggenauer and Jochen Wessel 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 Sebastian Röglinger.

We would also like to thank the Software Technology Research Laboratory (STRL) from the De Montfort University, especially François Siewe and Hussein Zedan, for providing the appropriate environment for research.

REFERENCES