Building CPU Stubs to Optimize CPU Bound Systems: An Application of Dynamic Performance Stubs

Peter Trapp, Markus Meyer, Christian Facchi
Institute of Applied Research
University of Applied Sciences
Ingolstadt, Germany
{*trapp, meyerma, facchi*}*@haw-ingolstadt.de*

Helge Janicke, François Siewe
Software Technology Research Laboratory
De Montfort University
Leicester, United Kingdom
{*heljanic, fsiewe*}*@dmu.ac.uk*

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 increased 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 evaluates a possibility to simulate the functional behavior of software algorithms by the use of the *simulated software functionality*. These can be used by the *dynamic performance stub* framework, e.g., to build a *CPU stub*, to replace the algorithm. Thus, it describes a methodology as well as an implementation and evaluates both in an industrial case study. Moreover, it presents an extension to the *CPU stubs* by applying these stubs to simulate multi-threaded applications. The extension is evaluated by a case study as well. We show show that the functionality of software algorithms can be replaced by *software simulation functions*. This stubbing approach can be used to create *dynamic performance stubs*, such as *CPU stubs*. Additionally, we show that the concept of *CPU stubs* can be applied to multi-threaded applications.**

Keywords-software performance optimization; CPU bound systems; simulated software functionality; stubs; multi-core; multi-threaded

I. INTRODUCTION

CPU stubs [1], which are a subset of *dynamic performance stubs* (DPS) have been introduced in [2]. They can be used for "hidden bottleneck" detection, and a cost-benefit analysis can be performed by demonstrating the level of optimization potential. This leads to more gain-oriented performance optimizations. These benefits are not addressed in other software performance engineering methods (see [3]–[7]). *DPS* extend these methods by simulating various levels of system load. Hence, the problem of a missing cost-benefit analysis can be bypassed. The *DPS* can be used within the software development cycle of large software systems, e.g., in telecommunication systems.

Many system architectures achieve 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 *DPS* can be used to optimize CPU bound processes by using *CPU stubs*.

A. Dynamic Performance Stubs

The idea behind *DPS* is a combination of performance improvements [3]–[7] in already existing modules or functions and the stubbing mechanism from software testing [8], [9]. The performance behavior of the component under study (CUS) will be determined and replaced by a *DPS*. This stub can be used to simulate different performance behaviors, which can be parameterized. Typically, the CUS is the part of the software under test (SUT) that has been identified as a potential performance bottleneck. The optimization expert can use *DPS* to analyze the performance of the SUT. This procedure relates to stubbing a single ("local") software unit, and a local stub has to be built. The *DPS* 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".

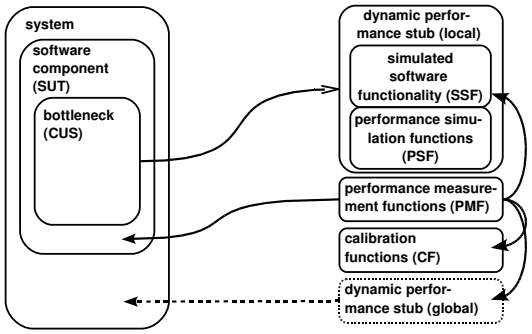


Figure 1. Interactions of "Dynamic Performance Stubs"

Figure 1 sketches the design and the interaction between a real system on the left and the *DPS* 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 *DPS* and will not really replace a software unit. In the context of *DPS*, the system under test (SUT) is a software module or function, which includes a software performance bottleneck.

The framework of the *DPS* consists of the following parts, which is presented in Figure 1:

- Simulated Software Functionality
  The *simulated software functionality* (SSF) is used to simulate the functional behavior of a CUS or a software performance bottleneck. This can be achieved by generating valid output values for dedicated input values without executing the original functionality. Another possibility is to simulate different states of objects inside of the CUS. Hence, the application can be executed without the original functionality as it is realized by the *SSF*.
- Performance Simulation Functions
  Performance simulation functions (PSF) provide the ability to simulate the performance behavior of the replaced CUS. They are divided into four categories, as also in [3], [10]:
  - 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.
- 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 *DPS*, the reader is referred to [2].

CPU Stubs: CPU stubs, as a special subset of *DPS*, can be used to handle CPU bound processes. These processes are highly utilizing the CPU so that the CPU is the bottleneck. Therefore, a general approach to parameterize the runtime behavior and CPU usage has been achieved and a possible implementation has been presented in [11]. Additionally, an extension to multi-core and parallel processing applications has been done in [1].

Memory Stubs: Memory stubs are separated into *cache memory-* and *main memory stubs*.

- *Cache Memory Stubs* can be used to simulate the data cache access behavior of software modules or functions to improve suspected memory bottlenecks. The algorithm, a validation as well as an evaluation by means of a proof of concept for cache memory stubs have been published in [12].
- *Main memory stubs* simulate the stack and heap behavior of software modules or functions. They are an extension of the *DPS* framework to simulate the main memory behavior to achieve a cost-benefit oriented optimization. They are defined in [13].

B. Content of the Paper

In Section II, the problem of simulating the results of software algorithms is addressed, by describing a novel approach to the simulation of software functionality using stubs. Starting with evaluating requirements concerning the simulation functions, a methodology is presented. Additionally, the concept of a possible implementation is depicted. Both, the methodology as well as the implementation are evaluated in an industrial case study to optimize a network component of a long term evolution (LTE) telecommunication system. This section extents and completes the approach of *CPU stubs* as published in [1].

In Section III, the paper shows an introduction to the *CPU stubs* as presented in [1]. Here, the *CPU PSF* are presented and a methodology for using *CPU stubs* to optimize CPU bound systems in multi-core or parallel processing environments is given. The introduction as well as the methodology depicts the concept of *CPU stubs* and have been published in [1].

In Section IV, the *CPU stubs* are extended to simulate the performance behavior of multi-threaded applications. This extension is realized by defining objectives and by presenting a novel approach for *multi-threaded CPU PSF*. Additionally, the approach is validated by a case study.

Finally, related work for the *DPS* and for the *SSF* is provided in Section V.

II. Simulated Software Functionality

The *SSF* allows the replacement of an existing software module or function by a stub, in order to do software performance improvement studies. In the context of this paper, the *SSF* is used to replace a software bottleneck (CUS) with a *DPS* being able to simulate different performance scenarios to estimate the benefit of potential performance optimizations.

In the following, the requirements to the *SSF* and the methodology is presented. A implementation of the *SSF* is given. This section is concluded by an industrial case study.

A. Requirements

In order to be able to replace a bottleneck with a *DPS*, it is necessary to recreate the functionality of the software module or function. Hence, the following requirements can be defined and subdivided into: *requirements on the system*, which have to be satisfied by the SUT and *requirements on the SSF*, i.e., how the *SSF* has to behave:

1) **Basic Requirements on the system:**
   a) *Deterministic CUS Behavior*
      The software module or function has to have a deterministic functional behavior. Any execution of the function with the same input values returns the same output values, e.g., deterministic output values depending on the input values. The performance behavior of the function has to be deterministic, too.
   b) *Reproducible test execution*
      The used test environment and test scenarios have to deliver reproducible results. This is a common requirement to any test environment.
   c) *Automated test case execution*
      It is preferable if the test cases can be executed automatically. This property significantly reduces the effort for repeated executions of the tests. Additionally, reproducible test scenarios can also be used for performance measurements.
2) **Requirements on the SSF:**
   a) *Automatic generation of the serialization specification*
      The *serialization specification* is a description, which provides the procedural method to serialize the data types used in the *SSF* library. This *serialization specification* shall be generated automatically, because it removes additional effort for the user of the *SSF* and decreases the amount of possible errors, e.g., writing a wrong *serialization specification*. Hence, a serialization functionality shall be provided as well as an almost automatically *serialization specification* shall be generated. These can be used to automatically store the C++ objects.
   b) *Record and restore C++ data structures*
      It has to be possible to record and restore C++ data structures. Especially, it has to be possible to record and restore classes including non-public members, structures and lists. Moreover, the *SSF* has to be able to work with "NULL"-pointers, e.g., the "NULL"-pointers shall be stored in the trace file and restored during the stubs execution.
   c) *Simulate the functional behavior*
      The *SSF* shall be able to restore the functionality of the CUS. Moreover, it has to be able to restore all recorded C++ data structures into the memory of the SUT. Additionally, it shall be able to create objects if they are not available in the system.
   d) *Simulate the functional behavior with appropriate performance*
      The *SSF* has to be able to restore the functionality in negligible time, which is at least faster than the execution of the original software function. This is necessary to reduce the runtime overhead during the execution. Hence, the performance parameters can be easily adjusted using the *PSF*. This requirement mainly applies if the *SSF* is used in the context of *DPS* performance measurements. In this case, the requirement has to be fulfilled.

Especially, the requirements to the system (Requirements 1a and 1b) as well as the Requirements 2b and 2c are important. Not reaching them renders the *SSF* unusable. Requirement 2d is mainly important in the context of *DPS* as this requirement enables the performance adjustments, which are necessary for the *DPS*' approaches. This requirement may not be that important if the *SSF* approach is used in different scenarios. The Requirement 2a can only be fulfilled partly as stated in Section II-C1. The Requirement 1c is only suggested as it can significantly remove the overhead for applying the *DPS* framework in the performance evaluation study.

Moreover, there are some requirements to the C++ compiler [14]. The compiler shall be deterministic, e.g., the compiler has to produce an identical memory layout of two isomorphic classes. Whereas, this can not be strictly guaranteed, it is very unlikely that a non-deterministic C++ compiler is standard-compliant [14]. The "g++" of the gnu compiler collection (GCC) [15], which is used for the evaluation in this paper, fulfills the requirements.

In the next subsection, the methodology for using the *SSF* is presented.

B. Methodology

The *DPS* methodology, identifies potential performance bottlenecks that are replaced by stubs to facilitate gain-oriented performance improvements. The functionality of the potential bottleneck (CUS) is recorded using the libSSF (see Section II-C). The system's behavior with respect to the CUS is analyzed by replaying the functionality using the *SSF* with varying performance measures.

An overview is presented in Figure 2. It describes the overall process to replace the bottleneck by a *DPS*. There are three different parts, which have to be done. First, a header file is generated, which includes the *serialization specification*. Second, the functional behavior of the bottleneck has to be recorded and stored in a trace file. Finally, the trace file can be used to simulate the functional behavior of the bottleneck. The steps as provided in this methodology are presented as circles including the step number.

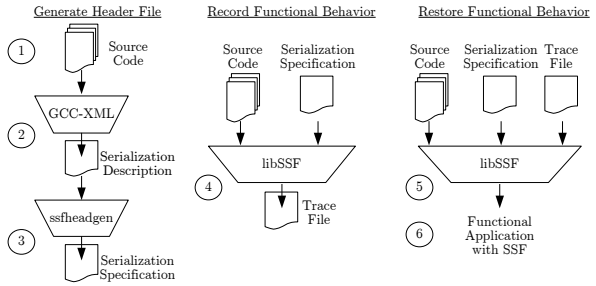


Figure 2. Overview of the *Simulated Software Functionality*

The methodology for the *SSF* approach consists of the following steps:

1) **Identify Serialization Objects:**
The *SSF* is able to store and restore different states of the traced objects. Thus, it can be used to simulate the results of several algorithms. In some cases, it is necessary to use parts of the original functionality to improve the simulation results. Here, the content of the object may be stored and restored before and after executing the original software functions.
2) **Create Serialization Description:**
In this step, the serialization description of the identified objects has to be created. This is simply done using the "GCC-XML" tool set [16].
3) **Create Serialization Specification:**
The *serialization specification* is created. It contains a description to serialize and de-serialize the objects which will be stubbed. The serialization objects (see Step 1) and their description (see Step 2) are processed by the *ssfheadgen* tool, which is a part of the *libSSF* library, to generate a C++ header file that contains the *serialization specification*. This specification has been created automatically for many basic data types, as explained in Section II-C1, but, can also be easily extended by the developer to support object serialization. This header file will be included into the CUS in the next step.
4) **Record the state of the objects:**
In this step, the CUS is adjusted to store the results of the algorithm using the *libSSF*. Furthermore, the test cases that utilize the functionality, which will be stubbed (see Step 1), have to be executed and the state of the results have to be recorded into a trace file.
5) **Create Functional Software Behavior:**
The original functionality, which is a part of the CUS, is replaced by the *SSF*. This is detailed in Section II-C.
6) **Test the instrumented CUS:**
As the stub has been created in Step 5, the functionality of the stub has to be validated. The instrumented CUS is validated against the previously recorded behavior of the CUS that contained the original functionality. As there is the potential of introducing functional errors during the instrumentation, the instrumented CUS is re-validated against the previously recorded execution trace. If the validation passes, the stubs can be used to do the performance study with the *DPS* framework.

This section has shown how a stub can be created using the *SSF*. The following section presents a possible implementation called *libSSF*.

C. Implementation

The implementation of the *SSF* is done in a library called *libSSF*. The library can be included into any C++ source code and allows for the storage of the content of C++ data structures into a binary trace file. Moreover, the *libSSF* can also be used to read from the trace file to reconstruct the C++ data structures. Thus, the functional behavior of the CUS is also recreated. For this reason, the source code of the application will be parsed using the "GCC-XML" tool set [16], which generates an XML description of a C++ program from GCC's internal representation. Based upon this serialization description, *libSSF* generates an internal representation of the objects that will be stubbed. The following functionalities are provided by the *libSSF*. These are the general steps (see Figure 2):

1) Generate Header File
This file includes the *serialization specification*.
2) Record Functional Behavior
This functionality will be used to store the results of the software functionality of the CUS.
3) Restore Functional Behavior
This functionality will be used to simulate the software functionality of the CUS.

Following, the listed items are described in more detail.

1) Generate Header File (Serialization Specification): A tool provided by the *libSSF*, called "ssfheadgen", parses the XML description of the "GCC-XML" tool. It extracts the type information and generates a C++-header file, which includes the internal representation of the objects that will be replaced by the *SSF*.

This header file can be included into the C++ source code of the CUS and contains the *serialization specification* of the objects. Beside for the basic data structures, e.g., basic data types or fixed size arrays, which have to be serialized and de-serialized, the developer has to adjust the header file to his needs. This has to be done manually as it is not always possible to determine the size of data associated with a pointer value.

Whenever possible, the header file already contains comments and suggestions to assist the developer in serializing the object, e.g., for pointers or arrays[1]. Moreover, the header file includes the original names, as used in the CUS source

[1]In these cases a "stop criterion" has to be specified by the developer.

```
template <> void Stubfactory::serializeType(class array_class *ssfSaveObj) {
  void *prt = ssfSaveObj;
  struct ssfSave_array_class *ssfObject = (ssfSave_array_class *) ptr;
  this->serializeArray(ssfObject->ac_i, numberOfElements);
}
```

Listing 1. Example: Serialization of a Fixed Sized Integer Array inside of a C++ class

code, of the replaced objects, so that the developer can reuse these names for convenience.

Listing 1 shows an example of the serialization of an array of integers ("ac_i"). The array is a private member of a class ("struct array_class"). This snippet is used to deserialize as well as to serialize the data values of the object.

The "serializeType"-function from Line 1 will be called indirectly inside of the CUS. The provided parameter specifies a C++ class which shall be serialized and stored. In Line 2, a type cast of the object pointer to a void pointer is done. This is necessary for being able to furthermore cast the pointer to a "struct", which reflects the C++ class. In this case, the private or protected members of the provided class ("ssfSaveObj") can be accessed and, hence, stored. This is done in Line 4, where, the private member, which is a fixed size array in this example, will be copied into the trace file.

This example shows that private and protected members can be serialized. Other serialization functions are available to support the developer.

2) Record Functional Behavior (Binary Format): The C++-header file, which has been generated by "ssfheadgen", is included into the CUS. And, the software tests, which have been done to identify the serialization objects, have to be repeated. Now, the information of the objects are stored in a trace file. This recording of the data structures is done using the function "saveStateOfParam", which is included into the CUS. This function is provided by the *libSSF*. The declaration of the function can be seen in Listing 2.

```
template <class TYPE> void saveStateOfParam(
    const char *name, const char *type, TYPE
    *dataVar);
```

Listing 2. Stores a Data Structure e.g. class

The following three parameters have to be passed to the "saveStateOfParam"-function call:

1) "const char *name": This is the name used to store the object in the trace file, e.g., "conn".
2) "const char *type": This refers to the type and name of the object to be stored, e.g., "class Connection".
3) "TYPE *dataVar": This is a pointer to the data which will be stored, e.g., the value of the conn variable.

The three given examples in the list above can be interpreted as: Store the value of the *conn* variable, which has an object type *"class Connection"* into the trace file using the name *"conn"*. The function uses the parameters and stores the data structure as well as additional information into a binary trace file. The structure of a trace file entry is given and described below:

- Test Run
  This is an internal reference counter starting from zero. The "test run" number can be used to summarize different stored variables into a combined run, e.g., if the value of a variable has to be stored before and after some modification within a single execution of the function.
- Size of Object Name (Byte)
  The size of the object name is given in bytes including a "NULL"-termination character.
- Name of the Object
  The name of the object which has been stored. It is usually the same name as the name of the object within the original source code and can be used for referencing the stored data.
- Size of the Object Type Name (Byte)
  The size of the object type name is given in bytes including a "NULL"-termination character.
- Name of the Object Type
  The name of the object type, e.g., "class Connection", which means the data entry refers to a C++ class named "Connection". Here, object type refers to any C++ data structure and can also be a basic data type such as an integer.
- Additional Information
  The "additional information" (addInfo), which is a field of 8 bits, is used to determine whether a fully initialized object has been stored or if a "NULL"-pointer has been passed. This information is stored in the first bit flag. The remaining bits of this field are unused. Hence, the first bit of "additional information" field is set to "0" if an initialized data structure has been stored. In this case, the following two additional data fields are stored in the trace file for this test run:
  - Size of the Stored Data (Byte)
    This is the size of the serialized data in bytes.
  - Stored Data
    These are the values of the data structure. The data have been serialized in advance and are successively ordered in the trace file.

The information is stored in a binary format for performance

reasons. A decoded as well as semicolon separated example trace entry is given in Listing 3.

In this case, an object "conn" of the "class Connection" type has been stored. The values of the serialized private members are: "1", "2", "1" and "302845744".

```
1;5;conn;17;class Connection;0;
    14;1;2;1;302845744;
```

Listing 3. Example: Decoded and semicolon separated trace file entry

The *libSSF* provides an option to generate a trace file decoder for a dedicated trace file. This has been implemented to provide human-readable traces to the developer.

3) Restore Functional Behavior (Deserialization): The recorded values have to be recreated into the memory of the used C++ data structure. Hence, it is necessary to overwrite the values already stored in the memory of the object. To do this, three different cases have to be considered:

1) The object as well as trace data are available.
   In this case, the existing attributes of the object have to be overwritten as the object is already available in the system.
2) The object does not exist but data are available.
   The object has to be created and initialized using the values of the trace file. Moreover, the pointer to the object has to be returned to the system. This is possible as the delivered memory pointer has to be "NULL". In this case, no memory is associated with the original object. As there is no reference available, dangling pointers can not occur.
3) The object does not exist and no data are available.
   This case happens if an initialized object is not necessary, e.g., if the return value of a search algorithm does not find the item. I.e., the CUS returns a "NULL" value. In this case, the object pointer passed to the "loadStateOfParam"-function of the *libSSF* (see Listing 4) has to be "NULL". Here, no memory will be allocated.

A fourth case is that an initialized object has been passed to the *libSSF* but no data are associated within this test run. In this case, a "NULL" pointer would have been returned by the *libSSF*, which will overwrite the original pointer value of the object. This is not allowed as it would cause a memory leak. Additionally, it is not possible to delete the associated object data as this could lead to a double free error. Hence, the developer has to care about this particular case, e.g., deleting the object and setting its pointer value to "NULL" before the "restore" is function called.

The restore functionality of the *libSSF* is implemented by the "loadStateOfParam"-function call. This function will be used to replace the software functionality of the CUS. The declaration of the function is given in Listing 4.

The parameters passed to the *libSSF* are as follows:

```
template <class TYPE> TYPE* loadStateOfParam(
    string dataVar, TYPE *dst);
```

Listing 4. Restore Functionality of the *libSSF*

- "string dataVar": This is the name of the object, which will be deserialized. Here, the same name as specified as the first parameter of Listing 2 is used, e.g., if "conn" is passed to the "loadStateOfParam"-function, the with conn associated data will be returned.
- "TYPE *dst": This is a pointer to an object which will be overwritten by the values read from the trace file. Hence, it is implemented as a template any type of the object can be deserialized and restored by the *libSSF*, e.g., the "class Connection" with an instance name "conn" can be used.

In the case that an object has to be created within the *libSSF*, the pointer value of the newly allocated memory will be returned to the CUS. Here, the original value of the pointer will be overwritten so that the allocated memory can be deleted inside of the original software.

The provided methodology and implementation will be applied to a real world example which is presented in the following section.

D. Case Study

The *DPS* framework has been used to optimize several algorithms of a long term evolution (LTE [17]) telecommunication system.

This section describes the application of the methodology and the newly developed *SSF* within a performance improvement study. The main contribution of this case study is to show that the software functionality of the CUS can be replaced by the *SSF*. This includes the following steps:

- The *serialization specification* is generated.
- The software functionality of the CUS can be recorded.
- The software functionality of the CUS can be replaced by the *SSF*. In this case, the SUT shall be fully functional for this particular test scenarios.

Last but not least, the case study provides performance measurements to validate that the *libSSF* can be used in the context of the *DPS* framework that will be used to evaluate software performance optimization potentials.

1) Test Environment: The measurements have been done in a test environment. The used platform is based on an Intel Xeon CPU, which is an IA-64 architecture and includes OS and memory.

The application has been built using the available build system of the company. This uses the "g++" of "GCC" (Version 3.4.3) for host test environment evaluations. The "-Os" compiler option has been used, which is basically a "-O2" but without optimization flags that increases the code size.

As the presented measurements have been done in a test environment, the results can only be used for validation purposes of the *SSF* but do not reflect the performance of the telecommunication system.

The requirements to the software and test environment, as specified in Section II-A, for using *DPS* are fully met. These are, in particular, a deterministic CUS as well as an automatic and reproducible test case execution environment.

2) Application of the Methodology: The SUT has a "ConnectionContainer" class which stores several connections of the type "class Connection". The function "get(connID)" returns the connection specified by the connection identification ("connID") which is an object of the "Connection" class. Moreover, it returns "NULL" if the connection does not exist in the "ConnectionContainer". The connection class has four private members as can be seen in Listing 5.

```
1  class Connection
2  {
3     ...
4     private:
5        TL3ConnectionId     m_connectionId;
6        u16                 m_streamId;
7        TUeContextId        m_contextId;
8        TAaSysComSicad      m_uecAddress;
9     ...
10 }
```

Listing 5. Excerpt of the Class "Connection"

Step 1: The "get(connID)" function has been identified as bottleneck and, hence, the "Connection" class has been chosen for serialization.

Steps 2 & 3: In the next step, the members of the "Connection" class are serialized using the "GCC-XML" tool set (Step 2). An example serialization output of the *ssfheadgen* (Step 3) is shown in Listing 6.

```
1 name= 'm_connectionId' id= _4096
2     type= _1501
3 -> name= 'TL3ConnectionId' id= _1501
4         type= _1532
5   -> name= 'u32' id= _1532
6           type= _73
7     -> name= 'unsigned int' id= _73
8             type=
```

Listing 6. Example of Serialized Class Member

Here, only the first member "m_connectionId" is presented. The "GCC-XML" combined with the *ssfheadgen* tool identified the "m_connectionId" over four serialization steps as an unsigned integer.

As of Step 3, the *serialization specification* is written into a C++ header file. The first part of the file contains the serialized object, which is presented in Listing 7. As can be seen, the "Connection" class, which has been converted into a data structure, consists of four "private" members, which are integers.

```
struct ssfSave_Connection{                          1
  unsigned int          m_connectionId;             2
  short unsigned int    m_streamId;                 3
  unsigned int          m_contextId;                4
  unsigned int          m_uecAddress;               5
};                                                  6
```

Listing 7. Serialized "Connection" Object

The second part, which is the serialization code, is also included into the file. An extract is shown in Listing 8 for this case study.

```
template <> void Stubfactory::serializeType(        1
    class Connection *ssfSaveObj) {
  void *ptr = ssfSaveObj;                           2
  struct ssfSave_Connection *ssfObject = (          3
      struct ssfSave_Connection *) ptr;
  this->serializeType(&ssfObject->                  4
      m_connectionId);
  this->serializeType(&ssfObject->m_streamId);      5
  this->serializeType(&ssfObject->m_contextId)      6
      ;
  this->serializeType(&ssfObject->m_uecAddress      7
      );
}                                                   8
```

Listing 8. *Serialization Specification* of the "Connection" Object

Here, the "serializeType"-function in Line 1 is able to serialize a object of the "Connection" class. It calls internally several different "serialize"-functions (Lines 4 - 7), which overload the function from Line 1. The "serialize"-functions from Lines 4 - 7 call internally a "serializeAtom"-function, which is able to store and restore basic data types. The values of the variables are stored in their associated members of the data structure (see Listing 7). The "type casts" in Lines 2 and 3 are necessary to access the private members of the "Connection" class (see Section II-C1).

Step 4: Now as the setup has been finished, the measurements have to be repeated to store the software functionality of the CUS. The chosen test case is a functional test case which evaluates different use case scenarios. We only studied a small subset of the test case for the *libSSF*. In our context the test case includes 40 times calling the stubbed functionality ("get(connID)"-function). The test case includes the following use cases: "create new object", "reuse existing object" and "delete and create new object". A decoded excerpt of the recorded trace file is shown in Listing 9. Lines 4-7 of the listing show the recorded values of the private members of the "Connection" class for the second test run.

Steps 5 & 6: In the last two steps, the stub has to be created using the restore functionality. Moreover, the proper

```
1 testrun:0; sizeObjectName:5; objectName:conn;
      sizeObjectType:17; objectType:class
    Connection; addInfo:1;
2 testrun:1; sizeObjectName:5; objectName:conn;
      sizeObjectType:17; objectType:class
    Connection; addInfo:0;
3   sizeOfData:14;
4   m_connectionId:1;
5   m_streamId:2;
6   m_contextId:1;
7   m_uecAddress:302845744;
```

Listing 9. Excerpt of a Decoded Trace File

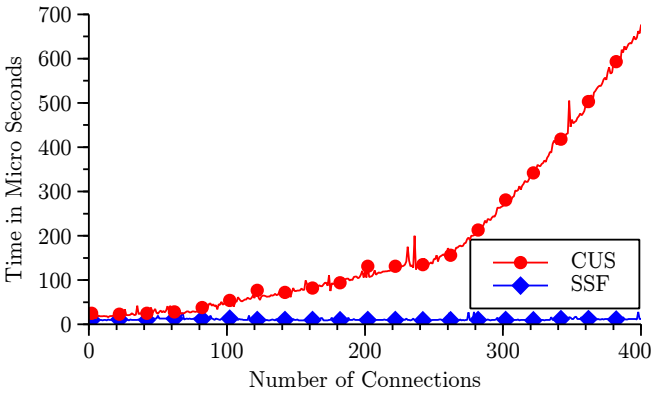


Figure 3. Compare the Times Between Original and Stubbed Software Functionality

working of the stub has to be validated.

The functionality, which will be replaced by the stub, is removed from the CUS and replaced by the restore functionality of the *libSSF* in order to simulate the original functionality. Now, the test case, as used in Step 4, is executed and the results are validated. In the test environment the test case passed. In this case study, the *libSSF* was able to simulate the functional behavior of the CUS.

3) Performance Measurements: The concept of the *SSF* will be used within the *DPS* framework to simulate different performance behaviors of a software bottleneck. Hence, the time to restore the functional behavior of the CUS is critical.

A case study using the *DPS* for optimizing CPU bound processes has been presented in [18]. The focus was to optimize a CPU bottleneck. The case study in this section uses the measurement results of [18], but, interprets the results from a different point of view.

Here, the differences between the execution time of the CUS and the execution time for the *SSF* have been evaluated. In the case study of [18], a previous version of the *libSSF* has been used. However, the results of [18] can still be used for this evaluation as only smaller changes have been done.

The same test environment and software functionality, as described in [18] has been used. A description of the environment and software functionality can also be found in Section II-D1. The chosen test case started with a single database entry and ramped up to searching 400 database entries. Each test case has been done five times and a statistical evaluation was performed by evaluating the minimum, average, maximum and the squared coefficient of variation (SCV, see [19]). The time values have been recorded in cycles using the time stamp counter (TSC, see [20]), which has been read by inline assembler.

In Figure 3, the time behavior for searching an entry in the database (y-axis) depending on the amount of database entries (x-axis) is presented. The lower line (diamond) shows the results for restoring the functional behavior of the search algorithm by using the *libSSF*. The upper line (circle) depicts the original behavior of the CUS.

The new evaluation pointed out the average time for restoring the functional behavior of the "get(connID)" function is 11 µs[2], which is approximately 30800 cycles. The SCV is 0.0135. This factor indicates that it takes approximately always 11 µs without significant variations to simulate the functionality independent of the amount of database entries.

In contrast, the original functionality to identify the connection identification number ("connID") took a minimum of 22µs if only a single entry was in the database and about 675µs if 400 entries have to be searched. The measured results show an exponential increase in time for an increasing database size.

As can be seen, the *SSF* was even in the worst case as twice as fast. Due to this, the identification of the software optimization potential and the improvement of the bottleneck's time behavior were easily realized. Moreover, the methodology of using *CPU stubs* has been applied to the SUT in [18], successfully.

E. Discussion

The *SSF* can be used to store and recreate the functional behavior of software modules or functions of C++ applications. This implemented by the possibility to store and restore the values of C++ data types, e.g., data structures or classes including their private and protected members. Moreover, it is possible to use the *SSF* with applications which use many different programming techniques, such as virtual or abstract classes, inheritance or polymorphism. The functionality is implemented by a library called *libSSF* which can be included into the C++ source of the application.

Advantages: Using the *libSSF* has several advantages for the developer. The main topics are:

- **Store and Restore the Software Functionality**
  The *libSSF* can be used to record and restore the states of traced objects. Hence, it can be used to simulate

[2]The first message has been ignored to avoid side-effects that only occur for the first message ("first message effect").

software functionalities and algorithms, e.g., search-, sorting-, or calculation algorithms. Moreover, existing objects can be modified to the needs of the developer.

- **Mainly Automatic Header Generation**
  The header, which is generated by the *ssfheadgen* tool, can be easily included into the source code of the CUS. Here, only some small modifications have to be done. Moreover, the *ssfheadgen* tool provides suggestions to support the developer by this task. This enables the developer to easily trace and evaluate the content of C++ data types.
- **Reuse Object Names**
  Data types can be stored into and read from the trace file reusing the same names as in the original source code. This significantly reduces the complexity to use the *libSSF* within the CUS.
- **Using Data Types Multiple Times**
  The same variables can be recorded multiple times, even within one single execution of the CUS. Moreover, several different data types can be combined into dedicated runs as well as many different runs can be combined. This provides high flexibility for clustering different runs and data types for a better abstract view on the stubbed components.
- **Readable Values of the Objects**
  The *libSSF* provides the possibility to decode the binary trace files into human readable trace files. Hence, the values of recorded data types can be used for evaluating the outcome of algorithms and, hence, as additional debugging possibility.
- **Only Small Adjustments to the System**
  To simulate the software functionality, only smaller adjustments to the CUS have to be done. For recording, only the library has to be included as well as the necessary function calls have to be added. For restoring, additionally, the original functionality has to be removed, e.g., by uncommenting.

Restrictions: As often, there is a trade off between time and memory usage. If the library is used to restore the functionality of the software it will read the whole trace file into the memory during the initialization. Hence, it uses a lot of memory. Moreover, if a data type has been often recorded, each traced value is preloaded into the memory, e.g., if an integer has been stored ten times the *libSSF* will allocate ten times the size of the integer. This behavior has been chosen as the main focus is on the execution time of the restore functionality. It can be changed with some smaller modifications to the library to only load the data when they are needed. This leads to a longer execution time, of course.

Summary: As can be seen, the methodology of the *SSF* as well as their implementation, realized by the *libSSF*, can be used to record and restore the software functionality. Moreover, the time measurements of the *libSSF* have shown that it can be used in the context of the *DPS* framework.

This is an important contribution to the gain-oriented performance improvement framework *DPS*, as it allows to gauge the system-wide impact of a potential improvement before investing in the actual optimization of the algorithms that underly the functionality that has been simulated by the *DPS*. This informs decision making as to what bottlenecks should be prioritized and to what degree their optimization has a system wide impact.

The requirements on the system, which are 1a, 1b and 1c, as well as to the *SSF* (2b, 2c and 2d) as stated in Section II-A have been fulfilled with the *libSSF*. Finally, the Requirement 2a has been fully fulfilled in this particular case study, but, this can not be applied in a general way as explained in Section II-C1. However, this does not lower the contribution as the *libSSF* supports the possibility to manually adjust the serialization functions. And, hence, provides a broad range for applying the *SSF* to software systems.

III. CPU Stubs

This section introduces *CPU stubs*. They can be used to simulate the CPU performance behavior of a bottleneck. First, the *CPU PSF* are described. Afterwards, a methodology to apply the *CPU stubs* to multi-core and parallel processing is presented.

A. CPU Performance Simulation Functions

The *CPU PSF* 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 [21], this can be seen as “Running”. In this case, the CPU has to execute instructions. An implementation can be done by executing a busy loop [11], which executes the NOP instruction.
- 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 [21] and is simulated by delaying the execution of the process, e.g., using a sleep function call.

By using the *system influencing* and *system non-influencing CPU PSF* any state, regarding the CPU, of a process can be rebuilt.

B. Methodology

In this section, we revisit the methodology of using *CPU stubs* for simulating the execution states running and blocked/ready (see [21]) of individual processes that was presented in [1], [11] and show how these can be used in combination with the *SSF* that was presented in Section II.

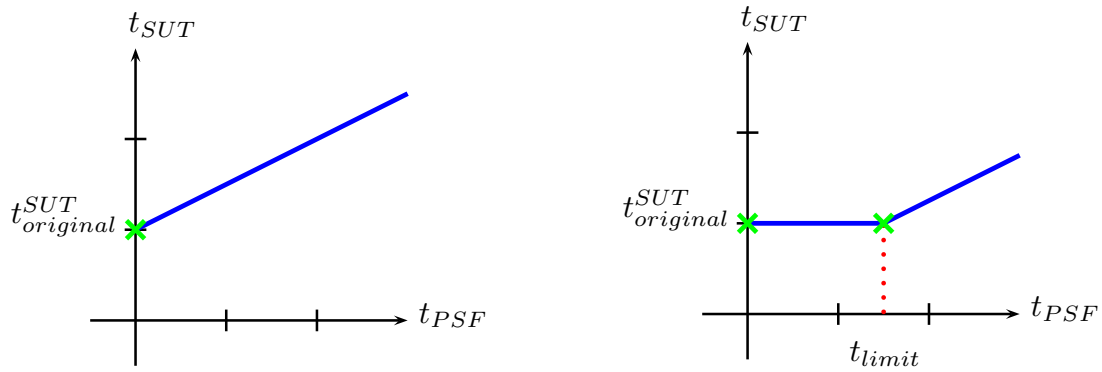
In the preceding section, we described a methodology and an implementation of simulating the software functionality of a CUS in order to determine the potential gain of optimization. Not all bottlenecks can be analyzed this way. With the parallelization of processing tasks by modern architectures and operating systems, concurrency issues and analysis of individual CPU usage becomes increasingly important. For this reason, we adapt our previous approach for the simulation of CPU behavior to a multi-core setting and show how the use of *CPU stubs* can complement the analysis of CUS using *SSF* with respect to concurrency.

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) [22], [23], e.g., profiling or tracing. Now, several performance indicators have to be determined:
   - t^{CUS}: Time spent in the bottleneck (CUS).
   - t^{SUT}: Time spent in the software module or function (SUT) from which the CUS is a part.
   - t^{CUS}_{busy}: Time spent in the CUS using the CPU. It includes the user-mode time as well as the system-mode time, see [21].
   - $t^{CUS}_{waiting}$: Time spent in the CUS waiting to be scheduled, see also: process state "Ready" in [21].
   - $t^{CUS}_{blocked}$: Time spent in the CUS waiting for an event, see also: process state "Blocked" in [21].

   The measured values have to be deterministic within several performance test runs.
2) **Validate CPU Bottleneck**
Here, a simple validation of the chosen CUS will be done. 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_{PSF}). The measured time of the SUT mainly follows one of the diagrams given in Figure 4.
In Figure 4a 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 4b 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 potential CPU bottleneck has to be identified (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 $t^{CUS}_{blocked}$, as determined in Step 1, will be used to evaluate the CPU utilization of the CUS.
A value of $t^{CUS}_{blocked} = 0$ 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 $t^{CUS}_{blocked} > 0$ 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 *DPS*, which only simulates the functional behavior of the CUS using the *SSF*. 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^{STUB}_{flat}: Time spent in the *flat CPU stub*.
   - t^{SUT}_{flat}: Time spent in the SUT including the *flat CPU stub*.

   An indicator of the possible optimization amount can be evaluated by calculating:
   - $t^{CUS}_{reduced} = t^{CUS} - t^{STUB}_{flat}$
   - $t^{SUT}_{reduced} = t^{SUT} - t^{SUT}_{flat}$

   $t^{CUS}_{reduced}$ is the time, which has been reduced in the CUS. The $t^{SUT}_{reduced}$ 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, $t^{CUS}_{reduced} * iter$ and $t^{SUT}_{reduced}$ has to be compared. It is possible to use the factor $iter$ because the same CUS is executed, so each iteration takes the same amount of time. The values t^{CUS} and t^{SUT} are taken from Step 1. Now, the calculated values can be compared and the following cases can be evaluated:
   - $t^{CUS}_{reduced} = t^{SUT}_{reduced}$: 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



(a) CUS seems to be a bottleneck

(b) CUS is no bottleneck

Figure 4. Validate CUS as a Bottleneck

in [24], 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.

- $t^{CUS}_{reduced} > t^{SUT}_{reduced}$: 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 and we can move on to the next step. 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. This is particularly the case if a change over in the critical path has happened (see [25]).

The case $t^{CUS}_{reduced} < t^{SUT}_{reduced}$ does not has to be considered. This would mean that the speed up of the execution time in the SUT is more than has been reduced in the CUS. Hence, the execution time of $t^{SUT} - t^{CUS}$ would has been decreased, but, the software within this part of the SUT has not been changed.

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

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}_{busy}). Hence, the following equation holds $t^{CUS}_{busy} = t^{STUB}_{idle}$. Where, t^{STUB}_{idle} 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 than 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.

The case that $t^{SUT}_{idle} > t^{SUT}$ does not have to be considered as it means that reducing the amount of instructions would lead to a longer execution time. This is not possible in typical CPU bound systems.

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, the amount of instructions determines the total execution time. Using a faster CPU means that each cycles and, hence, an instruction, is executed faster.

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_{busy}^{CUS}). Typically, the time spent in the PSF will be increased by 10% of the total execution time for each iteration. This can also be redone if a particalur time slice, e.g., between 20% and 30%, identified a change over in the critical path as explained in this step. The following values have to be measured:

- t_{busy}^{STUB}: Time spent in the *busy CPU stub*.
- t_{busy}^{SUT}: 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 5. 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_{min}^{SUT}). 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 chances 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_{min}^{SUT}).

This information can be used to identify "hidden" bottlenecks, e.g., a "hidden" bottleneck appears at t_{limit} of Figure 5. This limit is basically the maximum,

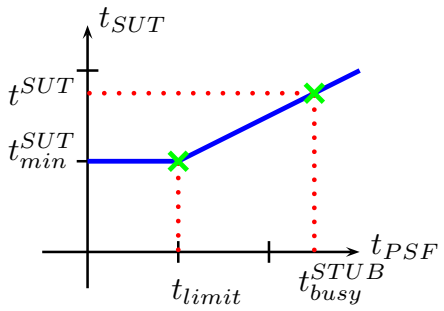


Figure 5. Limited Bottleneck

which can be achieved by an optimization of the CUS. Hence, it can be compared to a changeover in the critical path (see [25]). 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.

A CPU bound process can be optimized by using the described methodology. This is especially true for multi-core systems and multi-threaded applications where the bottleneck is single-threaded. Here, a changeover in the critical path after an optimization can be identified before doing the optimization itself.

C. Summary

In this section, we have presented the gain-oriented performance optimization methodology that can be used to optimize CPU bound bottlenecks. Increasingly, single-threaded systems are parallelized and software functions that constitute bottlenecks in the system use multiple threads that can be executed over a number of CPUs.

The following section provides an extension of the *CPU stubs* to simulate the performance behavior of multi-threaded bottlenecks.

IV. Multi-Threaded CPU PSF

As shown, using *CPU stubs* provide many possibilities to simulate the performance behavior of a system. The combination of *SSF* (see Section II) and *PSF* (see Section III) enable the execution of several performance simulations in order to determine the system's performance gain that can be achieved by its optimization.

To rebuild the system's functional behavior, the *SSF* was introduced in Section II. It is possible to generate the

required functionality half-automated. However, the amount of work that has to be done in order to be able to execute the desired simulations can still be rather high. In some cases, e.g., the generation of files, it can be more difficult to rebuild the functional behavior of a small piece of code than of larger software modules. In the "file" example above, it would be easier to use the created file as a whole within the *SSF*.

To be able to do this, the performance behavior of more complex software modules has to be rebuilt by the *CPU PSF*. For that the *CPU PSF* are extended to simulate the performance behavior of components under study that are even multi-threaded in this section.

A. Objectives

This section presents the main reasons for simulating a system's multi-threaded performance behavior using *CPU PSF*:

- In some cases, it is necessary to rebuild the performance behavior for more complex and even multi-threaded software modules. As shown above, the construction of the *simulated software functions* can be quite difficult for small functions. For that the possibility to simulate multi-threaded modules within the system and, thus, simplify and fasten the use of *CPU stubs* can extend the *CPU stubs*' field of application.
- In addition to that the methodology presented in III-B should be used in complex environments where the identification of performance bottlenecks can really be difficult. Here, *CPU stubs* can show their benefits when investigating the impact of a multi-threaded component onto the system's performance.

As presented, it is necessary to extend the available *CPU stubs* from [1] to be able to simulate the detailed performance behavior of more complex software modules, e.g., modules, which are multi-threaded. This extension is described in the following subsections. Next, the approach is evaluated.

B. Approach

To rebuild the performance behavior of complex, multi-threaded software modules, a defined interface is introduced to enable code generation. The data structured this way can be used to generate code that can be executed for the simulation. This interface uses four different actions to describe the behavior of a multi-threaded application. They can be separated into actions that are "specific for the *CPU stub's* performance simulation" and into actions that "model the creation and termination of threads" within the system.

- Describe the threads' performance behavior
  - **RUN** The *RUN* action is used to describe situations where CPU is used by the thread. This action is initialized by the keyword 'run' and followed by a number presenting the duration of this action in microseconds.
  - **SLEEP** This action is used to simulate the time while the thread does not use the CPU. This occurs when the process is blocked due to system calls or user interaction. The keyword for the *SLEEP* action is 'sleep' that is also followed by the time in microseconds.
    The blocked time, caused in the thread by waiting for another thread to terminate, e.g., waiting for a *JOIN*, is not simulated using the *SLEEP* action. If the addressed thread is still running and did not terminate until the call of the *JOIN* action, the calling thread is automatically blocked by the system. As this is the desired behavior for the simulation, it will not be simulated with a *SLEEP* action.
    Moreover, the blocked times are sometimes very short, e.g., a few microseconds. Thus, they can not be simulated accurate enough. Therefore, these short blocked time slices are not modeled via *SLEEP* actions but included into the *RUN* actions.
- Describe the threads' creation and interaction
  - **CREATE** In order to simulate multi-threaded software modules, it is necessary to create new threads. This is done by using the *CREATE* action. Its structure is 'create NUMBER', where NUMBER is used to identify the thread that has to be created. This is needed to be able to start the correct *PSF* for this thread.
  - **JOIN** To synchronize the created threads, the *JOIN* action is introduced. It consists of the keyword 'join' and the NUMBER representing the thread that has to be joined. As described in the *SLEEP* action, a call of *JOIN* blocks the calling thread until the addressed thread terminates. For that reason, a call of the *JOIN* action can result in an amount of time where the thread is blocked, even if this is not specially modeled within the interface.

For each of the threads that are rebuilt, an own file to describe its performance behavior has to be build in our environment. For that the code generation will produce one *PSF* for each thread. Those reference each other by using *CREATE* and *JOIN* actions.

The next section presents one appropriate implementation of the *multi-threaded CPU PSF*. Measurements are applied to obtain the data needed for the described interface.

C. Implementation

In order to rebuild the original performance behavior of a multi-threaded CUS and, therefore, to provide the data for the previously described interface, some measurements have to be done, to gather the needed amounts of execution time

and the information about the splitting and joining threads within the CUS.

1) Measurements: The Linux Trace Toolkit (LTTng) [26] is applied to collect all the information by using the available kernel markers. Based on scheduling events, the processes' execution time can be calculated. The markers for process creation and termination are used to determine the various threads that are spawned during the execution of the CUS and their respective timestamps.

2) Performance Simulation Actions: The *PSF* for the *RUN* and *SLEEP* actions are build as shown in [11]. The *RUN* action is generated as a *system influencing CPU PSF*, whereas, the *SLEEP* action is simulated by a *system non-influencing CPU PSF*. With the evaluation of the LTTng traces, the combination of busy and idle times can be simulated according to the original behavior.

3) Thread Behavior Actions: The actions *CREATE* and *JOIN* are used to describe the behavior of the threads that are simulated. In this approach, this is done by using POSIX threads [27]. When creating a new thread, its corresponding *PSF* is started. As described in the interface's structure, the *JOIN* action is used to synchronize the threads. This enables the *CPU stubs* to recreate the same thread behavior as the original software component does.

With the shown possibility to describe the performance behavior of a multi-threaded CUS, a case study can be performed to evaluate the approach's usability and the fact that multi-threaded performance behaviors can be rebuilt by *CPU PSF*.

D. Case Study

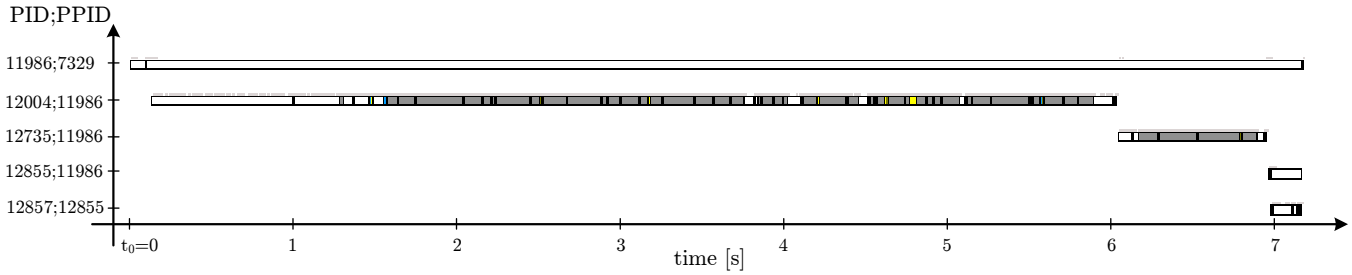
In this case study, the presented approach to rebuild complex performance behaviors of multi-threaded applications using *CPU stubs* should be validated. For that a compiling process of the GNU compiler (GCC version 4.6.0) is simulated.

1) Test Execution: The application runs on an Arch Linux operating system (kernel version 2.6.30.9) patched with the LTTng framework (version 0.160). The hardware is based on an Intel Centrino Core 2 Duo CPU with 2.26 GHz and has 4GB RAM. The calibration of the *CPU PSF* has been realized as described in [11].

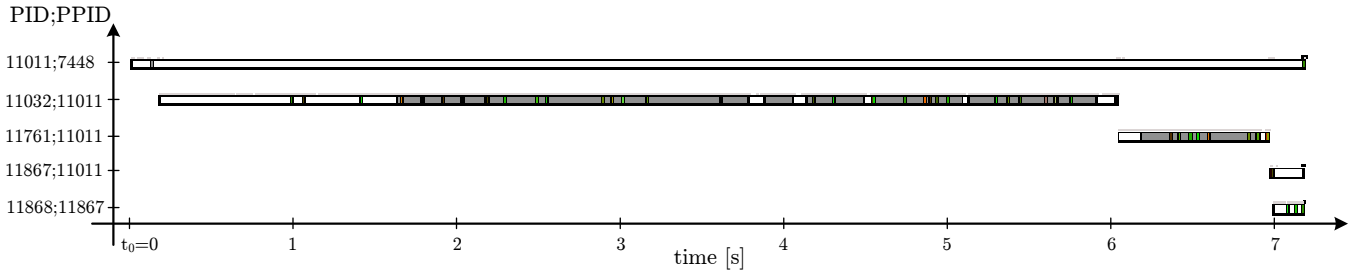
The test case chosen in this case study is the compile process to create a binary file using the GNU compiler (GCC version 4.6.0). In order to perform this case study a C-file containing an implementation of a quicksort algorithm is compiled. The compiled program is only of small size and does not have any further influence on this case study. This test case has been chosen as the GCC uses several threads depending on each other to compile the application.

2) Execution: This case study is performed in four major steps; record the data, generate the performance behavior, simulate the performance behavior and validation. These steps describe the process of creating the *CPU PSF*. The *PSF* can be used within the methodology, as shown in Section III-B, to simulate the performance behavior. E.g., the *CPU PSF* of a single thread can easily be adjusted to create an *idle CPU stubs*, which is Step 5 in the methodology for using *CPU stubs*.

1) **Record the Data**
   As a first step, the original performance data of the GCC call is recorded using the LTTng framework. This step corresponds to Step 1 of the methodology shown in Section III-B.
   Figure 6a shows the original performance behavior of the used GCC call. The x-axis presents the time in seconds. To get comparable results, the begin of the execution is set to $t_0 = 0$. The y-axis depicts the PID and the PPID of the created threads. The drawn bars present the performance behaviors of the single threads. When the CPU is used by the thread, the bar is gray, whereas, a white bar is used for the times where the threads are blocked or waiting for I/O. The other bars (black and colored) present other events that occurred during the execution of the system, such as paging and scheduling. As described in Section IV-B the time slices of those events are too short to be simulated accurate enough and, thus, are included into the *RUN* action.
   Regarding the PPID of each thread, Figure 6a also shows the threads' respective child threads and the order they are built. It can be seen that the thread with the PID 11986 (called Thread 11986) creates the Threads 12004, 12735 and 12855. Thread 12855 itself creates Thread 12857 before it joins back to its parent (Thread 11986).
2) **Generate the Performance Behavior**
   The data measured via LTTng has been transformed to fit to the described structure. As shown in Figure 6a the different process states are identified and rebuilt by performance behavior actions. The gray bars build *RUN* actions, whereas, the white bars are rebuilt by *SLEEP* actions. Additionally, the events that occurred when creating and synchronizing the threads are described by *CREATE* and *JOIN* actions.
   Listing 10 shows an excerpt of the interface's structure for the first created thread (Thread 11986) (see Figure 6a). Line 1 shows a call of the *SLEEP* action, which consumes 12780 microseconds. In Line 2 a call of the *RUN* action is carried out. The creation of threads is realized by *CREATE* actions as shown in Line 3. By the call of the *JOIN* action, e.g., in Line 7, the execution of the simulation blocks until the corresponding thread joined back. In this example, the numbers used for the creation and synchronization of the threads are the PID of the original threads. Beside the fact that this could be any number, it has been



(a) Original Performance Behavior



(b) Simulated Performance Behavior

Figure 6. Comparison of Original and Simulated Performance Behaviors of the GNU complier (GCC version 4.6.0)

chosen to be the PID of the thread for a simplified debugging in more complex environments.

```
1  sleep 12780
2  run 54
3  create 12004
4  run 3
5  sleep 11911
6  run 27
7  join 12004
8  run 74
9  sleep 9495
10 ...
```

Listing 10. Transformed Measured Data Used to Generate Simulation Code

For each of the five threads that run during the execution of the test, the code is generated as a combination of *system influencing CPU PSF*, *system non-influencing CPU PSF* and POSIX threads. Those pieces of code can be combined and used as the performance simulation code.

3) **Simulate and Validate the Performance Behavior of the *CPU stub***
After the *CPU PSF* have been built, the simulation is executed. To be able to compare the results, the measurements within the simulation were also done using the LTTng framework.
Figure 6b shows the simulated performance behavior of the system recorded by the LTTng framework. The x-axis presents the time in seconds and the y-axis depicts the PID and PPID of the executed threads. As in Figure 6a, the gray bars present the busy times and the white lines are used for the blocked times. In Figure 6b, it can be seen that the scheduling events (black and colored) are still triggered and thus are also recorded while the simulation is running. But events for paging that occurred during the original execution do not appear within the simulation. This is due to the *PSF*, as the *CPU stubs* shall only consume the CPU and shall not influence further elements of the system, e.g., the memory. Hence, it is the desired behavior that no page faults occur.

4) **Validation**
Figure 6 shows the comparison of the original (Figure 6a) and the simulated (Figure 6b) performance behavior of the GCC call. The evaluation of both graphs depicts that the performance of the GNU compiler call was rebuilt almost exact by the simulation. The running and blocked states of the single threads can be simulated accurately. The blocked times that occur during this simulation are a combination of the used *SLEEP* actions and the blocked time that occured due to the *JOIN* actions.
Minor inaccuracies of the simulation originate from the page faults that are not rebuilt by the simulation and the non-deterministic occurrence of the scheduling events. However, the performance behavior of the threads fits the values measured within the original execution of the system with high accuracy.

This simulation demonstrates that the *CPU stubs* can also be used to simulate the performance behavior of big software modules that may even be multi-threaded. This approach

can also lead to a simplified construction of the software functionality if the rebuild of a complex functional behavior is easier compared to the one of small modules, e.g., when generating whole files, as done by using the GNU compiler.

E. Summary

In this section, it has been shown that it is possible to rebuild the performance behavior of complex and multi-threaded software modules. An interface that can be used to describe the performance behavior of the system is introduced. Using this interface also simplifies the generation of code to simulate different performance optimization levels within the system by changing the performance parameters of the running and waiting times within the input data. Furthermore, the interface can be used in the previously described methodology and can lead to a faster and easier usage of *CPU stubs*. This uniform interface also increases the portability of *DPS* as it can be easily extended by code generators for further programming languages. Being able to recreate a complex performance behavior offers more possibilities when trying to detect the threads whose performance is critical for the system's performance.

The following section discusses the related work in the various areas that are affected by this paper.

V. Related Work

There are three major areas, which have to be considered for related work: *DPS*, *CPU stubs* and *SSF*.

Dynamic Performance Stubs: There are two different research scopes in simulation of the performance behavior. In [28] 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 [29] the usage of smart stubs for software analysis of functions and modules which are partly not available yet is described. Hence, the stubs simulate the budget regarding storage and time 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.

The *DPS* 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.

CPU stubs: The CPU executes the instructions of the applications. The scheduler exists to decide which process the CPU has to execute next (see [21], [30], [31]). It maintains several queues about the states of all processes: running, ready or blocked. If no process is either in the running or ready state the idle process is executed by the CPU.

From a process perspective, the process can be either executed by the CPU or is suspended. *CPU stubs* are targeting to simulate the time (CPU) behavior of a bottleneck. Hence, *CPU stubs* have to be able to switch the state. As only the scheduler decides whether a process is in the running or ready state, it is not possible to enforce a process to be in the running state in non-real time systems acting from user space side. Only the possibility to increase the chance to be scheduled soon into the running state exists, e.g., using priorities.

Hence, the *CPU stubs* have to simulate the remaining states "running" and "ready/blocked". The running state can be realized by a "do-nothing loop" and a state change can be initiated by delaying the process execution, e.g., using a sleep function.

In [32], [33] 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 *system influencing CPU PSF*, however, will be calibrated in an otherwise idle system with enough time for the OS. As experimentally proved in [11], in our environment the execution time of a process can be simulated with a do-nothing loop, predictably in contrast to [32], [33]. Additionally, because of the fact that such a loop has a defined number of instructions these loops can be used to simulate the time behavior of processes.

Simulated Software Functionality: A key functionality of the *DPS* is to record and to recreate functional behavior using the *SSF*. The recording requires the serialization of internal data-structures into a format from which they can be recovered at a later point. This functionality has been predominantly implemented in distributed systems where objects and code are marshaled for exchange between peers.

Most closely related to our serialization approach is the work in [14], [34] in which a "MPI Serializer" has been introduced. The target of this project is the efficiently and automated marshaling of C++ data structures. The tool generates automatically marshaling and unmarshaling code for the message passing interface (MPI), which is often used

as communication interface in high performance computing (HPC). The "MPI Serializer" is is based on the C++ serialization possibility of the "GCC-XML" project [16], which uses the gcc abstract semantic graph (ASG) scheme [35] to determine the *serialization specification*.

To some extent, our approach is similar to [14], [34] as both projects needs to serialize C++ data structures. However, it differs in many details. E.g., it has been decided to store and restore the functional behavior of software modules, which will be replaced by a stub. This can be used to remove a software bottleneck. In contrast, the focus in [14], [34] is to provide marshaling code for the message parsing interface.

However, both projects are based on the abstract semantic graph scheme provided by the "GCC-XML" project.

In [36], a lightweight fact extractor is presented. It utilizes XML tools, i.e., XPATH and XSLT, to extract static information from the C++ source code files. The approach is to transfer the source code into "srcML", which is a XML representation of the file. The fact extractor is mainly used to parse and search the source code. This technique is often used for reverse engineering, maintenance, testing or even in general development of software systems. This approach is based on "CPPX" [37], which is an open source C++ fact extractor. The fact base, which is generated by "CPPX", can be used as input for software development tools, such as integrated development environments (IDE). It enhances these tools' functionalities, for example by source code visualization, object recovery, restructuring and refactoring.

As in [14], [34], the approach of [36] highly differs from our approach, as it is not supposed to store and recreate the functional behavior of software modules. [36] mainly delivers a XML presentation of the extracted facts of the source code.

VI. Conclusion and Future Work

This paper evaluates our novel approach to the replay of functional behavior of software algorithms by the *SSF*. This functionality is used by *CPU stubs*, which are a subset of the *DPS* framework. These *CPU stubs* consist of the *CPU PSF*, which have been extended to simulate multi-threaded applications.

In order to achieve these results, two distinct functionalities have been combined: *SSF* and *CPU PSF*.

It has been shown that the functionality of software can be replaced by the *SSF* almost automatically. A methodology to use this functionality is given and a possible implementation is provided. This is concluded by an industrial case study.

Moreover, *CPU stubs* can be used to simulate the performance behavior of complex software modules. This can simplify the creation of *software simulation functions* and can be used to determine the impact of multi-threaded components onto the system's performance. The usability of the introduced interface, to describe the multi-threaded performance behavior, has been shown in a case study.

The following aspects concerning the *SSF* and *CPU stubs* will be addressed in the future work:

- *SSF*
  - Evaluate the memory influence of the *libSSF* and identify improvement possibilities to reduce this overhead.
  - The *libSSF* will be extended to easily trace more different data structures. Especially, it should be improved to stub "STL lists"and "STL vectors".
  - By now, the *libSSF* is based on some workarounds. Further possibilities to access private and protected members of classes will be evaluated, e.g., by using friend classes. Moreover, a redesign will be done.
- *CPU Stubs*
  - Consideration of further events within the interface, to rebuild the communication behavior of the threads more accurate.
  - Definition and evaluation of a methodology on the creation of *multi-threaded CPU PSF*.

Additionally, the methodology of *CPU stubs* will be evaluated by complex and multi-threaded case studies.

We have shown that *CPU stubs* can be used to simulate the functional as well as performance behavior of a CUS. This can be used to evaluate performance optimization potential depending on the system. Hence, it is possible to identify "hidden" bottlenecks as well as further improvement possibilities. Moreover, a gain-oriented performance analysis can be achieved.

Acknowledgment

This research was supported by a long term evolution system developing company. The authors would like to thank the LTE group 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 and the reviewers.

References

[1] P. Trapp, M. Meyer, and C. Facchi, "Using CPU Stubs to Optimize Parallel Processing Tasks: An Application of Dynamic Performance Stubs," in *ICSEA '10: Proceedings of the International Conference on Software Engineering Advances*. IEEE Computer Society, 2010.

[2] P. Trapp and C. Facchi, "Performance Improvement Using Dynamic Performance Stubs," Fachhochschule Ingolstadt, Tech. Rep. 14, Aug. 2007.

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

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

[5] D. J. Lilja, *Measuring Computer Performance: A Practioner's Guide*. New York: Cambridge University Press, 2000.

[6] N. H. Gunther, *The Practical Performance Analyst*. McGraw-Hill Education, 1998.

[7] J. J. Marciniak, *Encyclopedia of Software Engineering*, 2nd ed. John Wiley & Sons Inc, 2002.

[8] A. Bertolino and E. Marchetti, *Software Engineering: The Development Process - A Brief Essay on Software Testing*, 3rd ed. John Wiley & Sons, Inc., 2005, vol. 1, ch. 7, pp. 393–411.

[9] I. Sommerville, *Software Engineering*, 6th ed. Pearson Studium, 2001.

[10] J. Hughes, "Performance Engineering throughout the System Life Cycle," SES Inc., Tech. Rep., 1998.

[11] P. Trapp and C. Facchi, "How to Handle CPU Bound Systems: A Specialization of Dynamic Performance Stubs to CPU Stubs," in *CMG '08: International Conference Proceedings*, 2008, pp. 343 – 353.

[12] P. Trapp, C. Facchi, and S. Bittl, "The Concept of Memory Stubs as a Specialization of Dynamic Performance Stubs to Simulate Memory Access Behavior," in *CMG '09: International Conference Proceedings*. Computer Measurement Group, 2009.

[13] P. Trapp and C. Facchi, "Main Memory Stubs to Simulate Heap and Stack Memory Behavior," in *CMG '10: International Conference Proceedings*. Computer Measurement Group, 2010.

[14] W. Tansey and E. Tilevich, "Efficient automated marshaling of C++ data structures for MPI applications," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, apr. 2008, pp. 1–12.

[15] "GCC, the GNU Compiler Collection," online: http://gcc.gnu.org/, June 2011, [July 5, 2007].

[16] GCC-XML, "GCC-XML, the XML output extension to GCC!" online: http://www.gccxml.org/, 2010, [July 5, 2011].

[17] F. Khan, *LTE for 4G Mobile Broadband: Air Interface Technologies and Performance*, 1st ed. Cambridge University Press, 2009.

[18] P. Trapp, C. Facchi, and M. Meyer, "Echtzeitverhalten durch die Verwendung von CPU Stubs: Eine Erweiterung von Dynamic Performance Stubs," in *Workshop "Echtzeit 2009 - Software-intensive verteilte Echtzeitsysteme", Informatik Aktuell*. Springer Verlag, 2009, pp. 119–128.

[19] R. Srinivasan and O. Lubeck, "MonteSim: A Monte Carlo Performance Model for In-order Microarchitectures," *ACM SIGARCH Computer Architectur News*, vol. 33, no. 5, pp. 75–80, Dec. 2005.

[20] Y. Etsion and D. Feitelson, "Time Stamp Counters Library - Measurements with Nano Seconds Resolution," The Hebrew University of Jerusalem, Tech. Rep. 2000-36, 2000.

[21] A. S. Tanenbaum, *Modern Operating Systems*, 2nd ed. Prentice-Hall, Inc., 2001.

[22] C. U. Smith, "Formal Methods for Performance Evaluation," in *Introduction to Software Performance Engineering: Origins and Outstanding Problems*, 2007, pp. 395 – 428.

[23] C. U. Smith and L. G. Williams, "Best Practices for Software Performance Engineering," in *Int. CMG Conference*, 2003, pp. 83–92.

[24] S. McConnell, *Code Complete: A Practical Handbook of Software Construction*, 2nd ed. Redmond, WA, USA: Microsoft Press, 2004.

[25] S. Chapman, "Finding the Critical Path - A Simple Approach," in *CMG '09: International Conference Proceedings*. Computer Measurement Group, 2009.

[26] "Linux Trace Toolkit Next Generation," online: http://www.lttng.org, 2011, [July 5, 2011].

[27] "The Open Group Base Specifications Issue 6 IEEE Std 1003.1," online: http://pubs.opengroup.org/onlinepubs/009695399/basedefs/pthread.h.html, 2004, [July 5, 2011].

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

[29] D. J. Reifer, "The Smart Stub as a Software Management Tool," *SIGSOFT Softw. Eng. Notes*, vol. 1, no. 2, 1976.

[30] D. P. Bovet and M. Cesati, *Understanding the LINUX KERNEL*, 3rd ed. O'Reilly, 2005.

[31] A. Fugmann, "Scheduling Algorithms for Linux," Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, DTU, 2002.

[32] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications," in *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*. New York, NY, USA: ACM Press, 2005, pp. 303–312.

[33] D. Tsafrir, "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*. New York, NY, USA: ACM Press, 2007, p. 4.

[34] W. Tansey, "Automated Adaptive Software Maintenance: A Methodology and Its Applications," Master's thesis, Virginia Tech, May 2008.

[35] N. A. Kraft, B. A. Malloy, and J. F. Power, "A tool chain for reverse engineering c++ applications," *Sci. Comput. Program.*, vol. 69, no. 1-3, pp. 3–13, 2007.

[36] M. Collard, H. Kagdi, and J. Maletic, "An XML-based Lightweight C++ Fact Extractor," in *Program Comprehension, 2003. 11th IEEE International Workshop on*, May 2003, pp. 134 – 143.

[37] CPPX, "CPPX - Open Source C++ Fact Extractor," online: http://www.swag.uwaterloo.ca/cppx, july 5, 2011.