Abstract

Dynamic performance stubs provide a framework for the simulation of the performance behavior of software modules and functions, allowing many to be used as an extension to software performance engineering methodologies. The methodology of dynamic performance stubs can also be used for gain oriented performance improvements and it is also possible to identify “hidden” bottlenecks and to prioritize optimization possibilities.

Main memory stubs are defined to improve the simulation possibilities of the dynamic performance stubs framework. They are able to simulate the heap and stack behavior of software modules or functions. This paper verifies the usability of the given method and provides an algorithm to use this functionality in real-world scenarios. Additionally, the algorithm is evaluated by means of a proof of concept. Moreover, steps towards using main memory stubs are presented. We have shown that main memory stubs can be used to simulate the heap and stack behavior.

Key words: software performance optimization, main memory simulation, dynamic performance stubs

1 Introduction

Dynamic performance stubs have been introduced in [TRA07]. 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. This leads to more gain-oriented performance optimizations.

In the past, performance increases in many system architectures have been achieved through higher CPU speeds, and more recently, through using multiple cores. Yet, the memory speed, and hence the memory access times, did not increase to the same order as the CPUs frequencies [SEA00]. This has led to the fact that many current systems are heavily memory bound and consequently software performance optimization studies are often targeting to the improvement of the memory usage. The methodology of dynamic performance stubs can be used to optimize these memory bound systems by main memory stubs.

1.1 Dynamic Performance Stubs

The idea behind dynamic performance stubs is a combination of performance improvements [JAI91, GUN98] in existing modules or functions and the stubbing mechanism known from software testing [BER05, SOM01]. The performance behavior of the component under study (CUS) will be determined and replaced by a software stub. This stub can be used to simulate different performance behaviors which can be parameterized, and 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 and hence it will be called “local”. Thus, a “local stub” has to be built. The performance simulation functions (PSF) 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 only a single software component. This stub will be called a “global stub”.

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 arrow...
head indicates a replacement. Filled arrowheads describe the extension of an 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, and are divided into four categories:

  - CPU
  - Memory
  - I/O
  - Network

  Furthermore, memory PSF will be subdivided into the cache memory PSF and main memory PSF respectively their according stubs, i.e., cache memory- and main memory stubs.

- 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 [TRA07]. A short introduction to CPU stubs and memory stubs is given below.

**CPU Stubs**

CPU stubs are targeting 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 order to create these CPU stubs, a methodology for evaluating and creating these stubs is provided in [TRA08]. An application of the CPU stubs to achieve real-time behavior is available in [TRA09b]. Here the methodology of CPU stubs has been used to improve the performance behavior of the LTE\(^1\) telecommunication software from Nokia Siemens Networks (NSN). Furthermore, the usability of CPU stubs has been extended to support multi-core and parallel processing applications in [TRA10].

**Memory Stubs**

Memory stubs are now furthermore separated into cache memory- and main memory stubs. Therefore, the originally known memory stubs are renamed to cache memory stubs.

**Cache Memory Stubs**

The 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 [TRA09a].

**Main Memory Stubs**

Main memory stubs simulate the stack and heap behavior of software modules or functions. They are an extension of the dynamic performance stubs framework to simulate the main memory behavior to achieve a cost-benefit oriented optimization. They are defined and evaluated in this paper.

1.2 Content of the Paper

This paper describes the concept of main memory stubs in more detail. It mainly discusses a possible implementation of the performance simulation functions and evaluates the usability to simulate the stack and heap access behavior. Additionally, it evaluates the main memory performance simulation functions by means of a proof of concept. Moreover, the first steps towards a methodology for using main memory stubs is presented.

2 Basics

This section is based on a literature review, which briefly describes related work. Furthermore, it provides some basic understanding of the memory layout of ap-

---

\(^1\)LTE: Long term evolution is the successor of UMTS.
2.1 Related Work

Performance skeletons, as described in [SOD08], are used to simulate the performance behavior of applications. This has been achieved by capturing the execution behavior and creating synthetic program skeletons. Hence, the skeletons are using an instruction mix, which is similar to the instructions used by the application under study. The main target is to simulate the performance influence of the application in long running scenarios by short time execution of the skeletons. This can be used to accurately estimate the performance in heterogeneous and shared computational grids. In [SUB08] the automatic constructions of these skeletons is shortly described. The possibility to replicate the memory performance behavior is studied in [TOO04].

The approach of the dynamic performance stubs framework significantly differs from the performance skeleton approach. The performance skeletons are used to simulate the performance behavior of all performance bounds concurrently, e.g., cache and main memory, and only aim on the reduction of the running time of the application. In our approach, we simulate each performance bound independently, to be able to separately adjust each performance behavior to the needs of the performance improvement study. This leads to a gain-oriented optimization. Moreover, the dynamic performance stubs framework is able to simulate a function or module. Whereas, the performance skeletons are always simulating the behavior of the whole application. Therefore, the approach of the performance skeletons can not be applied to the dynamic performance stubs framework to simulate the memory behavior.

2.2 Memory Usage in Computer Systems

An application typically consists of five memory segments [SIL05]: Code, Data, Block Started by Symbol (BSS), Heap and Stack.

The code segment, also known as text (segment), is a portion of memory, which stores the instructions executed by the application. The next two segments, i.e., data and bss, stores variables, which are allocated during the compile time. The heap segment stores variables, which are dynamically allocated during run-time of the process. The stack memory is used to store temporarily used variables, e.g., within function calls. Hence, the stack memory is more often allocated and freed than the heap memory. Additionally, the stack usually allocates only few bytes, whereas, the amount of bytes allocated on the heap is higher. Moreover, the stack only de- and allocates the data on top of the memory, whereas, the heap memory always tries to allocate the data in an appropriate memory region. Hence, fragmentation of the heap can happen [SIL05, KER88], which can lead to “unnecessary” memory allocations.

The shared libraries, which are used by the process, are typically stored between the heap and the stack segment [SIL05].

Normally, the segments are ordered as described, starting from the lower addresses to the higher, but, this can differ in various architectures. The order can be seen in Linux-based operating systems (OS) using the process file system (procfs, see manual page of the procfs), e.g., in /proc/PID/maps.

The memory layout, as described above, applies to the virtual address space, also known as logical address space, of applications. The memory management unit (MMU) translates the virtual addresses to real addresses [SIL05]. This translation is often assisted by the translation lookaside buffer (TLB) [TAN01].

The main memory will be allocated by processes in pages, which are successive memory chunks with a size of “pagesize”² [SIL05].

If a process allocates main memory, exceeding the available memory already fetched, a page fault will be created by the OS [SIL05]. Two different types of page faults can happen: Minor- and major page fault.

A minor page fault, also known as soft page fault, happens if the newly allocated page has to be requested from the main memory. If the memory has to be fetched later from a higher level memory, e.g., hard disk drive (HDD), a major page fault, also known as hard page fault, is triggered by the OS. Minor page faults are less expensive in terms of the time than major page faults. The amount of page faults of a process can be read through the “getrusage()” function³ or through the procfs, e.g., in /proc/PID/stat.

2.3 Memory Handling in Applications

This section discusses several assets and drawbacks of the different memory allocation functions. More information about the explained function calls can be found in the corresponding manual pages.

Initializing Memory Memory, which is allocated, will usually be filled with data. The “memset()” function can be used to initialize the requested memory to a predefined value. The time needed for initialization significantly depends on the amount of memory.

Allocating Stack Memory Usually, the stack is not handled by the programmer in a direct way. It serves as a highly dynamically memory for storing temporal used data. Nevertheless, the stack can be allocated,

²The pagesize can be evaluated in POSIX-based OS’s using “sysconf(_SC_PAGESIZE)” from unistd.h
³The function can be accessed through sys/resource.h in Linux-based OS.
e.g., using the “alloca()” function. The implementation is very fast on most systems, as it is only adjusting the stack pointer register. As described in Section 2.2, the stack can not be fragmented. As drawback, an allocation failure is not indicated and, therefore, it is often handled as the “out-of-stack” space situation, e.g., with a segmentation fault. Moreover, it is recommended to avoid the “alloca()” function with large unbounded allocations.

### Allocating Heap Memory

The heap memory is designed to provide a flexible run-time storage to the programmer. Whenever heap memory has been allocated, it has to be freed, e.g., using the “free()” function call, to avoid memory leaks. Further problems using the heap memory can exist, e.g., dangling pointers or freeing the same memory twice [McC04].

Common heap allocation functions are: “malloc()”, “realloc()” and “memalign()”. “Malloc()” fetches main memory in multiples of system page sizes but uses only the requested memory. The remaining heap memory can then be used later on. The requested system pages do not have to be continuous in the main memory. Freeing the allocated memory can lead to heap fragmentation as described in Section 2.2. “Realloc()” is similar to “malloc()” but can be used to resize the memory as requested by the programmer. Moreover, “realloc()” acts like malloc if a NULL-pointer is given and can be used as “free()” if the SIZE parameter is omitted. “Memalign()” basically uses the “malloc()” function and then aligns the obtained value. Moreover, the memory returned by “malloc()” is usually aligned, anyway. Thus, the use of the “memalign()” function is deprecated anyway. Not mentioned for the sake of completeness.

Another possibility to allocate heap memory is to use the “calloc()” function. “Calloc()” is designed to allocate memory for arrays. Here, the number of elements as well as the size per element can be specified. Additionally, “calloc()” initializes it is allocated elements to zero.

The available heap memory is usually handled by an ordered list [KER88]. Hence, the “malloc()” function call can be expensive if the heap is highly fragmented. Moreover, the heap has to be reconfigured if memory is newly allocated or freed by the process. Thus, using the heap memory is usually more expensive regarding the execution time than using the stack memory. However, memory from the heap can be used throughout the process run time and, usually, more heap memory can be allocated than stack memory.

Another aspect of the heap memory is that “freed” heap memory is often not directly returned to the system. Moreover, it is stored in a malloc pool for further usage [WOL09, STE08].

### Validation Environment

As a validation environment, the test equipment of the Nokia Siemens Networks next generation Radio Network Controller (ngRNC) has been used to validate the measurements. It hosts a 2.8 GHz Intel Pentium 4 central processing unit with hyperthreading disabled. The operating system is a standard Linux running on a 2.6.22 kernel. The kernel was built tickless with the high resolution timers enabled. We ran the tests with executables generated by the GCC of version 4.2.1. In order to avoid unwanted optimizations by the compiler, optimization flags were not used for compiling the stub.

The single test runs are executed under the same test conditions. Each test has been executed several times in order to get statistically viable results. Hence, the minimum (min), average (mean), maximum (max) and squared coefficient of variation (sqd coeff of var) has been used to validate the test results.

The usability of the approach was validated twofold:

First, an execution driven evaluation has been done. Thus, the execution time is validated using the time stamp counter (TSC) (see also [ETS00]). Additionally, the amount of minor page faults is evaluated using the “getrusage()” function. Moreover, a binary analysis has been done, where necessary.

The second stage of evaluation is simulation based. Hence, the valgrind tool suite (www.valgrind.org), especially callgrind and massif, has been used. Callgrind is a callgraph and cache simulation tool. The results can be evaluated using KCachegrind (kcachegrind.sourceforge.net). Massif evaluates the stack and heap memory allocation behavior of processes by evaluating the allocation functions. More information on older versions of massif can be found in [RUN92].

The described environment and test procedures will be used for evaluating the main memory stubs throughout the whole paper.

### 4 Concept

This section evaluates different allocation and initialization possibilities as described in Section 2.3. Moreover, a new developed algorithm to use the allocated pages is described and evaluated. The validation- as well as the tracing environment, as described in Section 3, is used.

#### 4.1 Memory Handling

In this subsection a discussion of the behavior of the following functions is done: “alloca()”, “calloc()”, “realloc()”, “memset()” and “distmemset()”. The functions “calloc()” and “memalign()” are not discussed further.

---

Many different “malloc()” implementations exist, e.g., “dmalloc()” or “ptmalloc()” (see also [LEV00]).

Valgrind (massif) has been used in version 3.5.0.
Because, “alloca()” basically combines “malloc()” and “memset()”, which are studied separately.

The test cases have been executed five times but only the evaluation of the third execution is displayed, unless otherwise mentioned. This has been done to simplify the reporting as all five tests show similar results.

**Allocation Functions** An allocation function validates and ensures that the process will get enough memory as requested, if available. Hence, the heap allocation functions searches the already available heap memory for the amount of free space. If not enough space is available, it requests new memory from the system. This behavior is necessary as any heap memory can be freed during runtime, which leads to heap fragmentation. The stack memory allocation function basically only adjusts the stack pointer register.

Figure 2 compares the three mentioned allocation functions. The x-axis lists the number of the iteration done for allocating the 512 bytes of memory. The y-axis shows the time needed to execute the according function in terms of CPU cycles.

![Figure 2: Time spent in an Allocation Function Call](image)

As can be seen in Table 1, an already prefetched “alloca()” function call takes only 24 cycles on average. The first value of the test execution has been ignored because of the “first message effect”. The heap allocation needs 218 cycles for a “malloc()” respectively 279 cycles for a “realloc()” function call. The time spent in the heap allocation shows a minimum value, as it differs in other scenarios, e.g., for a fragmented heap.

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>mean</th>
<th>max</th>
<th>sqd coeff of var</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alloc #3</td>
<td>24</td>
<td>24</td>
<td>27</td>
<td>0.00053</td>
</tr>
<tr>
<td>Malloc #3</td>
<td>210</td>
<td>218</td>
<td>259</td>
<td>0.00103</td>
</tr>
<tr>
<td>Realoc #3</td>
<td>259</td>
<td>279</td>
<td>903</td>
<td>0.14116</td>
</tr>
</tbody>
</table>

Table 1: Time spent in an Allocation Function Call

Additionally, the measurements have shown that no page faults are triggered by the allocation function, e.g., we measured only one minor page fault instead of the requested 20 pages with the “alloca()” function call. This is especially true for requesting many pages at a single time. The pages are only fetched as soon as the data will be used. Hence, a memory set function together with the allocation function is used to create page faults.

**Memory Set Functions** As described above, allocation functions can not solely be used to create higher amounts of page faults in the system. Thus, the “memset()” function is used to initialize the memory, and therefore, to create the desired number of page faults. As with “memset()”, as each memory position will be overwritten we expected the time needed in the function to be directly linear to the amount of memory used. Moreover, changing the value of the initialization variable should not make any difference to the initialization time as the value should be available in one of the CPU registers. Besides of the timing evaluation, this estimation has been confirmed by an analysis of the binary.

![Figure 3: Different Memory Set Functions](image)

The time spent in the “memset()” function has been measured starting from initializing zero bytes to six pages. The pages have already been fetched and allocated in the process before measuring. Figure 3 displays the time needed to initialize one byte in cycles per byte on the y-axis. The x-axis shows the amount of allocated bytes. The dark-blue graph (diamond) illustrates the time spent in “memset()” using always the same initialization value “0” (memset static) and the purple graph (squares) for changing the value using the actual allocation amount (memset variable).

<table>
<thead>
<tr>
<th></th>
<th>min</th>
<th>mean</th>
<th>max</th>
<th>scov $^6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mem. St. #3</td>
<td>0.389</td>
<td>0.434</td>
<td>0.828</td>
<td>0.02289</td>
</tr>
<tr>
<td>Mem. Var. #3</td>
<td>0.385</td>
<td>0.441</td>
<td>0.704</td>
<td>0.01168</td>
</tr>
<tr>
<td>Distmem. #3</td>
<td>0.014</td>
<td>0.038</td>
<td>0.208</td>
<td>0.48015</td>
</tr>
</tbody>
</table>

Table 2: Evaluates different Memory Set Functions
Starting from approximately one page (4096 byte) the amount of cycles for setting a byte is almost constant. The average value is 0.434 cycles per byte for “memset static” and 0.441 for “memset variable”, as can be seen in Table 2 Lines 2 and 3. Their respective squared coefficient of variations are 0.02289 and 0.01168. The third test run results are displayed. The first three values of the tests have been ignored in Table 2 because of the initial overhead for small values.

It takes many cycles to set the memory using the “memset()” function, e.g., ∼10000 cycles for 6 pages with “memset static”. As we only need to write into each page once to create the necessary page fault, we used an algorithm, which is similar to “memset()” but a distance between the initialized bytes can be specified. The algorithm is called “distmemset()”. The distance is set to pagesize in the test runs, so a time efficient generation of page faults has been achieved. The results are displayed in Figure 3 and Table 2.

The time used for initializing one byte in this test environment takes 0.038 cycles on average, which reduces the number of cycles to ∼680 for six pages. We are aware that our algorithm is much slower than “memset()” if it is used with a distance of one byte. But, this does not matter as we only want to set one byte per page.

Another advantage of “distmemset()” compared to “memset()”, with our test conditions, is that, we only want to simulate the heap and stack behavior with the main memory stubs. The rest of the system should not be significantly influenced by the stubs, e.g., the main memory stubs should not create many cache events. As “memset()” sets every memory position a lot of cache events, e.g., level one cache misses, can be seen. The same execution setup has been used to measure the total amount of cache write misses. Callgrind shows 61339 for “memset static” compared to 1143 for “distmemset()”.

**Time to Serve a Page Fault** The following test has been executed using the “malloc()” and “distmemset()” functions. The time measured includes both function calls. Here, the time needed to allocate and load multiple pages to a process has been evaluated. The process starts using zero to 9900 pages with an increment size of 100 pages.

<table>
<thead>
<tr>
<th>min</th>
<th>mean</th>
<th>max</th>
<th>sqd coeff of var</th>
</tr>
</thead>
<tbody>
<tr>
<td>8068</td>
<td>8628</td>
<td>8757</td>
<td>0.00023</td>
</tr>
</tbody>
</table>

Table 3: Time Needed to Allocate and Load a Page

As can be seen in Table 3, the average time to load and set a single page is 8628 cycles. In this case, the “malloc()” and “distmemset()” functions have only small influences. The squared coefficient of variation reflects that the average value has only small variations. The first value, which is “allocate zero pages”, is not shown and evaluated in Table 3.

**Malloc Pool** As described in Section 2.3, memory that has been freed is not directly returned to the system but stored in a malloc pool. Hence, the process can reuse the memory if needed without requesting a new page from the system. The behavior of the malloc pool has been experimentally evaluated.

![Supposed and Measured Page Faults](image)

**Figure 4: Supposed and Measured Page Faults**

Figure 4 shows on the x-axis the number of each test run. The allocated size starts from zero to 114 pages with an increment size of six pages (20 test runs). The number of pages is listed on the y-axis. The blue graph (diamonds) shows the amount of allocation memory in pages. The purple line (squares) displays the number of minor page faults created by the process. As can be seen between test run two and twelve, much fewer minor page faults happen as pages are being requested. In this case, the pages from previous test runs are still available for the execution. Above 60 pages, which is ∼250000 bytes, the memory will be returned to the system.

![Time Influence of the Malloc Pool](image)

**Figure 5: Time Influence of the Malloc Pool**

The influence of the malloc pool can also be nicely displayed by the time needed to allocate and load a
Conclusion In this subsection different heap and stack allocation scenarios have been evaluated. As shown, each page has to be used at least once to really allocate the page to the process. Moreover, the time for different functions has been determined and the usability to simulate the memory behavior of applications has been shown. Hence, the functions provided can be used to create main memory stubs.

4.2 Algorithm
This subsection describes the algorithm that can be used to create main memory stubs. The requirements for the algorithm are:

- Allocate the heap or stack memory in chunks, i.e., do not always (de-)allocate the total amount.
- Allocate or free stack memory. Hence, the algorithm has to be recursive as only the return of a function call frees the stack memory.
- Allocate or free heap and stack memory in any order at a given time.
- The amount of allocated memory does not have to be changed at a given time.
- Use the allocated memory in order to create necessary page faults.

Listing 1 shows a possible implementation of an algorithm, which satisfies the requirements:

```c
extern struct memAlloc memUse[];
int totalHeap = 0;
int allocate(char **mHeap, int x, long int pagesize) {
    char *mStack = NULL;
    usleep(memUse[x].time);
    // allocate stack & distmemset
    // reallocate heap & distmemset
    do {
        if (x+1<NUMDATA && memUse[x+1].stackAlloc>0)
            x=allocate(mHeap, x+1, pagesize);
    } while (x+1<NUMDATA && memUse[x+1].stackAlloc>=0 && memUse[x].stackAlloc<0);
    x++;
    usleep(memUse[x].time);
    // reallocate heap & distmemset
    return x;
    // stack is freed
}
int main(int argc, char ** argv) {
    // get pagesize
    char *mHeap = NULL;
    int x=-1;
    while (x < NUMDATA - 1) {
        x=allocate(&mHeap, x+1, pagesize);
    }
    free(mHeap); mHeap=NULL;
    return EXIT_SUCCESS;
}
```

Listing 1: Algorithm to Specifically (De-)Allocate Heap and Stack Memory

-memory.

Stack Allocation To allocate new stack memory the “allocate()” function (Line 3) has to be called. The allocation of the stack memory is done in Line 6. Here, the “alloca()”- as well as the “distmemset()” function calls will be used to allocate the memory. The “alloca()” function call can not be done in an external function as the stack will be freed when the function terminates. In Line 20 the allocated stack will be freed. Hence, the “allocate()” function has to terminate.

Heap Allocation The “realloc()” function is used to resize the amount of allocated heap memory. The function is called together with “distmemset()” in Lines 7 and 17. Hence, it is possible to expand or shrink the heap memory at any given time.
Timing As the entry and exit of the “allocate()” function has to be used to change the stack memory, the time between two data samples has to be simulated at these points, too. Hence, a “usleep()” function is included in Lines 5 and 16.

4.3 Proof of Concept

A synthetic data set has been used to validate the usability of our algorithm. The input provides the possibility to validate every requirement given in Section 4.2. The test environment from Section 3 has been used. The graphs list on the x-axis the number of the sample points and on the y-axis either the amount of allocated memory in bytes (Figures 6 and 7) or the time in microseconds (Figure 8). All graphs are based on the same input data.

The purple graph (squares) in Figure 6 shows the amount of stack bytes allocated by the algorithm presented in Listing 1. The blue graph (diamonds) shows the amount of allocated stack in the process using the “stat” file of the procfs. The measured values are slightly above the specified. This is normal as, the total process stack size has been measured.

The first section (Samples 1 till 14) shows successive de- and allocations. Section II (Samples 15 till 21) presents alternating de- and allocations. For section III (Samples 22 till 25), the amount of allocated stack bytes is constant. Then, the function terminates (Samples 26 and 27).

Massif has been used to evaluate Figure 7. The blue line (diamonds) shows the massif results and the purple graph (squares) shows the supposed values by the data set. As it can be seen, some values, e.g., Sample 12 and 26 differ from each other. The reason is because of the massif tracing tool. Here, it is not possible to specify the time when a sample will be taken. Hence, the sample number can slightly differ.

Figure 8 shows the timing behavior of the evaluation. The blue line (diamonds) shows the time spent in the process measured with the “getrusage()” function. The values are above the predefined values (purple line (squares)), which have been specified in the data set. The difference is mainly because of the “distmemset()” function, which has been used twice, for heap and stack, per data sample. This behavior can be improved with the calibration functions as the time spent in the “distmemset()” function can be subtracted from the time in the “usleep()” function. This will be done as future work.

To conclude, the amount of heap and stack memory of a software module or function can be simulated. Additionally, all preconditions for the main memory stubs, as listed in Section 4.2, are met.

5 Steps Towards a Methodology

During the study of performance behavior of main memory bound systems, normally, a simple indicator that the system under study has a main memory bottleneck is initially given. In general two basic measurements can be used during the optimization phase:

Page Faults: Due to the fact that, an access of secondary memory is very time consuming, this should be avoided. Such an access will be in-
In the context of operating systems, page faults occur when a process requests access to a page of memory that is not currently in the system's memory. These faults are managed by the operating system, which loads the necessary pages from disk. This process can be costly in terms of both time and performance, especially if it becomes frequent or repeated. When this happens, it is referred to as page thrashing, which can significantly reduce the system's performance and ultimately its operational efficiency.

The number of page faults can be an indicator of whether a system is memory bound or not. If page faults are not measured, the real execution time can be chosen as the optimization criterion, and time has to be considered as a measurement. The following definitions are used:

- \( t_{\text{CUS}} \): Time spent in the bottleneck (CUS).
- \( t_{\text{SUT}} \): Time spent in the software module or function (SUT), which the CUS is part of.

In this methodology, the following steps are outlined:

1. **Determination of the Bottleneck**
   - The SUT has to be identified, and a suspected bottleneck (CUS) has to be located. This is typically done through software performance engineering (SPE) methods, such as profiling or tracing. According to the chosen measurement method, either the number of page faults or time constraints have to be determined. The measured values have to be statistically viable within several performance test runs.

2. **Create Main Memory Stub**
   - A main memory stub is created. The functionality of the software module or function has to be simulated using the **simulated software functionality**. Now, the main memory PSF will be inserted into the stub. They simulate the memory usage of the CUS. In addition, the time spent in the CUS \( t_{\text{CUS}} \), to guarantee the original timing behavior, has to be simulated by a CPU stub [TRA08, TRA09b]. Then, repeated measurements should deliver the same values as the original software.

3. **Validate Memory Bottleneck**
   - The next step is a modification of the memory usage parameter in the main memory PSF. First, the memory usage has to be increased, e.g., 5%, 10%, ..., 100%. If that leads to more successive page faults or a longer system time \( t_{\text{stubbled}} \), then the CUS is at least one main memory bottleneck. Please note that, the values for the initial page faults have to be subtracted as they will still occur in the background at this stage.

4. **Evaluate the Optimization Potential**
   - Now different measurements with different optimisation parameters have been realized, and the results noted for the next step, where a first simple cost estimation of the performance improvement can be carried out. Based on the results, the optimal cost-benefit ratio can then be determined.

5. **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.

**6 Conclusion and Future Work**

This paper evaluates the heap and stack behavior of processes. A study of the performance behavior of different memory allocation scenarios has been realized. An algorithm to simulate the heap and stack behavior of functions is described and evaluated by a proof of concept. Additionally, first steps towards a methodology for using main memory stubs during an optimization study are given.

We have shown that the heap and stack behavior of software modules or functions can be simulated. Thus, they can be used as main memory stubs, which are an extension of the dynamic performance stubs framework.

The following topics are considered as future work: Evaluate the **calibration functions** to reduce the “memset()” overhead. Furthermore, extend and improve the given methodology for using main memory stubs and study the heap fragmentation behavior of processes as well as the algorithm.

Additionally, the composition and aggregation of the different stubs will be studied. So a combination of different stubs can be achieved.

**7 Acknowledgment**

This research is granted by Nokia Siemens Networks (NSN). The authors would like to thank the LTE group in Ulm 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 Ian Baldwin, Markus Meyer and Sebastian Röglinger.

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


