



**How to Correctly Simulate
Memory Allocation Behavior of
Applications by Calibrating Main
Memory Stubs**

Working Papers

Arbeitsberichte

Peter Trapp, Markus Meyer and
Christian Facchi



**How to Correctly Simulate Memory
Allocation Behavior of Applications
by Calibrating Main Memory Stubs**

Peter Trapp, Markus Meyer
and Christian Facchi

Arbeitsberichte
Working Papers

Heft Nr. 20 aus der Reihe
„Arbeitsberichte – Working Papers“
ISSN 1612-6483
Ingolstadt, im Mai 2011

How to Correctly Simulate Memory Allocation Behavior of Applications by Calibrating Main Memory Stubs

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

March 2011

Abstract

Dynamic performance stubs provide a framework to simulate 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* targets to gain oriented performance improvement. Other applications include the identification of “hidden” bottlenecks and the prioritization of optimization alternatives. *Main memory stubs* have been developed to extend 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 extends and improves the simulation algorithm to be able to simulate constant stack values. Moreover, it presents calibration possibilities to improve the simulation results by determining the various overhead in the algorithm. The results are furthermore used to compensate inaccuracies in the simulation. Additionally, a proof of concept is given as validation of the results. This paper shows that, *main memory stubs* can be used to simulate the heap, stack and timing behavior exactly when considering the parameters determined by the *calibration functions*.

Keywords:Memory Systems; Software Performance, Evaluation and Testing; Modeling; Performance Optimization, Bounds, and Models; Case Studies

1 Introduction

Dynamic performance stubs have been introduced in [1]. They can be used for the detection of “hidden” bottlenecks. By demonstrating the optimization potential of the detected bottleneck a cost-benefit analysis can be performed, leading to a gain-oriented approach to performance optimizations.

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

1.1 Dynamic Performance Stubs

The idea behind *dynamic performance stubs* is a combination of performance improvements [2–4] in already existing modules or functions and the stubbing mechanism from software testing [5,6]. The performance behavior of the component under study (CUS) will be determined and replaced by a *dynamic performance stub*. This stub can be parameterized to simulate different performance behaviors. 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 *dynamic performance stubs* to analyze the performance of the SUT. This procedure relates to stubbing a single software unit. Hence, it will be called “local”. Therefore, a “local stub” has to be built. The *dynamic performance stub* can also be used to change the behavior of the complete system. A software module has to be created, which interacts “globally” in the sense of influencing the whole system instead of a single software component. This stub will be called a “global stub”.

Figure 1 sketches the design and the interaction between a real system on the left and the *dynamic performance stubs* on the right side. The unfilled arrowhead indicates a replacement. Filled arrowheads describe the extension of a unit by this feature and the dashed block provides an additional functionality to the *dynamic performance stub* and will not really replace a software unit. In the context of *dynamic performance stubs*, the system under test is a software module or function, which includes a software performance bottleneck.

The framework of the *dynamic performance stub* consists of the following

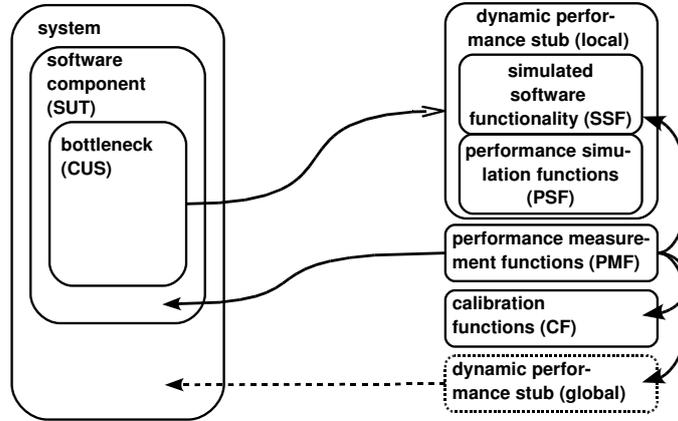


Figure 1: Interactions of “Dynamic Performance Stubs”

parts, which is presented in Figure 1:

Simulated Software Functionality (SSF). The *simulated software functionality* is used to simulate the functional behavior of a component under study 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 component under study. Hence, the application can be executed without the original functionality as it is realized by the *simulated software functionality*.

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” and “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 wrapper functions for the measurement functions already provided by the system.

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

For more detailed information on *dynamic performance stubs* the reader is referred to [1]. 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 implemented. The methodology of *CPU stubs* has been used to improve the performance behavior of a long term evolution (LTE) telecommunication software. Furthermore, the applicability of *CPU stubs* has been extended to support multi-core and parallel processing applications in [7].

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 [8]¹.

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 in [9].

1.2 Content of the Paper

The first part of this paper enhances the algorithm to simulate the memory behavior of an application, known from [9]. A closer view on the design and the execution of the algorithm is shown. An extension of the algorithm to recreate situations where the amount of allocated stack memory remains constant is introduced.

Second, in [9], there are some imperfections with the simulation of the time and memory allocation behavior. This paper extends the concept of *main memory stubs* by evaluating *calibration functions*. These can be used to adjust the *main memory stubs* to the system. This highly improves the simulation results for the “heap”, “stack” and “timing” behavior. Additionally, a proof of concept is given.

2 Main Memory Stubs

Main memory stubs are used to simulate the main memory performance behavior of a component under study in the context of *dynamic performance stubs*.

¹*Cache memory stubs* are referred to this in the earlier publication as *memory stubs*.

2.1 Methodology

To simulate the main memory behavior of applications the following steps have to be done. First, a main memory performance bottleneck has to be identified. Now, the bottleneck has to be evaluated, especially, the functional as well as the main memory behavior have to be determined. Afterwards, the functional behavior has to be rebuilt using the *simulated software functionality*. Moreover, the main memory *performance simulation functions* have to be recreated. This leads to a *main memory stub*.

Now, the memory behavior of the stub can be changed according to the needs of the performance study. Hence, several possible optimization levels as well as their influences to the system can be simulated. Moreover, studying the results can identify “hidden” bottlenecks, e.g., the timing behavior of a related software function can change because of the *main memory stub*. So, possible results of an optimization can be simulated before the optimization has to be done.

2.2 Performance Simulation Functions

In [9], *performance simulation functions* to simulate the main memory allocation of a SUT have been introduced. In this section, the algorithm is briefly presented.

Figure 2 shows the design of the memory simulation algorithm. The recursive function, needed to simulate the stack behavior, is called “allocate()”. This function is called when the amount of stack is increasing.

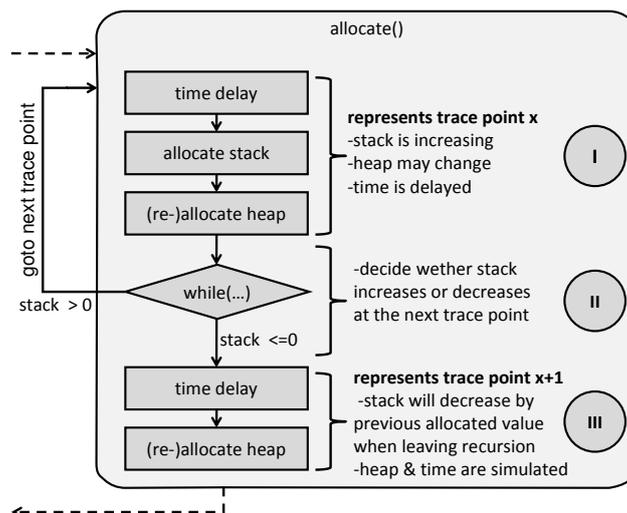


Figure 2: Memory Simulation Algorithm

Simulate Stack Allocation. In the first part of the algorithm (Part I in Figure 2), the time will be delayed as requested by the data set. Then, the stack memory is allocated by calling the `alloca()`-function and used with the `distmemset()`-function (see also [9]).

Next Trace Point. In the second part (Part II), the algorithm decides whether the stack is increasing or decreasing at the following trace point. It will again reach Part I if the stack is increasing. If the stack is shrinking, the recursion has to be left, which is realized in Part III.

Simulate Stack Deallocation. This is achieved in the third part (Part III). The time delay is simulated and the stack memory is automatically freed when leaving the `allocate()`-function.

After having left the `allocate()`-function, the algorithm is either back in the previous function, which initially called the `allocate()`-function, or it is in Part II and the `while(...)` condition proves once more if the amount of stack is increasing at the next trace point. This construction is needed to simulate a sequence of rising and trailing edges.

Simulate Heap (De-)Allocation. The heap memory can be de- as well as allocated in Part I and III. As it can increase and shrink at this points nothing special has to be considered.

2.3 Simulation Data File

To reduce the overhead created when running the algorithm, the measuring points used to simulate are written into a data structure within a header file that is used to compile the algorithm.

```

#define NUMDATA 4
1
2
struct memAlloc{
3
    int time;
4
    int stackAlloc;
5
    int heapAlloc;
6
} memUse [NUMDATA]={
7
    [0].time=129, [0].stackAlloc=300, [0].heapAlloc=0,
8
    [1].time=223, [1].stackAlloc=100, [1].heapAlloc=200,
9
    [2].time=384, [2].stackAlloc=-100, [2].heapAlloc=-40,
10
    [3].time=112, [3].stackAlloc=-300, [3].heapAlloc=-160
11
}
12

```

Listing 1: Example for a Simulation Data File

Listing 1 shows an example header file that can be used to build the simulation algorithm. For each measuring point, the data structure “memAlloc” contains the time elapsed since the previous point. Additionally, the change of heap as well as stack memory are contained. Every point is presented by one “memAlloc” struct. Those structs are inserted into an array of “NUMDATA” length. So the algorithm can easily switch to the next trace point and does not produce much overhead in execution time.

3 Improvement of the Algorithm

The algorithm presented has several drawbacks and insufficiencies which are addressed and improved in this section.

Insufficiency of the Algorithm. When taking a closer view to the algorithm, it becomes clear that the algorithm can not be used to simulate situations where the allocated amount of stack memory remains constant. The algorithm is designed to handle stack values not equal to zero. The implementation of the algorithm shown in [9] would interpret a constant stack value as a decrease. For that reason, it would leave the recursion and, so, free the amount of stack that was previously allocated. This behavior of the algorithm is insufficient to simulate the memory behavior of the component under test as exact as possible. Therefore, an adjustment of the algorithm is presented in this paper.

Improvement. In order to simulate situations in which the time proceeds and the amount of heap memory changes but the stack memory remains constant additional code has been developed.

```

inline int constantStack(char** mHeap, int x, long int 1
    pagesize, long int totalheap){
    while(x+1<NUMDATA && memUse[x+1].stackAlloc==0){ 2
        x++; 3
        usleep(memUse[x].time); 4
        *mHeap=allocateHeap(*mHeap, x, pagesize, totalheap) 5
        ;
    } 6
    return x; 7
} 8

```

Listing 2: Simulation of constant stack situations

Listing 2 shows a possible implementation to simulate trace points, where the allocated amount of stack does not change. This function checks whether

the next trace point has a constant stack value (Line 2). If it does, it increases the number of the trace point, represented by “x”, delays the time as desired and calls the `allocateHeap()`-function to simulate the heap behavior for this trace point (Lines 3-5). The value of stack is not changed. This sequence is repeated as long as the stack remains constant.

With help of this function, the algorithm can easily be adjusted. The possible situations, where a constant stack behavior can occur are:

- Between two rising edges
- Between a rising and a trailing edge
- Between two trailing edges
- Between a trailing and a rising edge

To cover all situations, the function has to be called on several position within the algorithm. The function has to check for the constant stack value after every rising edge as well as after every trailing edge.

```

int allocate(char** mHeap, int x, long int pagesize) { 1
    char* mStack = NULL; 2
    usleep(memUse[x].time); 3
    if (memUse[x].stackAlloc > 0) { 4
        mStack = (char*) alloca(memUse[x].stackAlloc); 5
        distmemset(mStack, 'a', memUse[x].stackAlloc, 6
            pagesize, totalstack);
    } 7
    *mHeap=allocateHeap(*mHeap, x, pagesize); 8
    x=constantStack(mHeap,x, pagesize); 9
    while (x+1<NUMDATA && memUse[x+1].stackAlloc >0){ 10
        x=allocate(mHeap,x+1, pagesize); 11
        x=constantStack(mHeap,x, pagesize, totalheap); 12
    } 13
    x++; 14
    usleep(memUse[x].time); 15
    *mHeap=allocateHeap(*mHeap, x, pagesize, totalheap); 16
    return x; 17
} 18

```

Listing 3: Improved implementation of the algorithm

Listing 3 shows the final implementation of the improved algorithm to simulate the memory behavior of the component under test. The `constantStack()`-function is placed after the rising edge (Line 9) on the one hand and after the end of a recursive call of the `allocate()`-function that is reached when the stack memory is decreased (Line 12). Additionally, the `distmemset()`-function has been adjusted to only use the newly allocated memory. This is achieved by passing the total memory size to the function.

Through these improvements of the algorithm, the simulation of the memory behavior has been adjusted. Because of this, situations in which the stack memory remains constant can now be simulated as well as the overhead in the `distmemset()`-function is reduced.

4 Calibration Functions

This section presents the evaluation of different overhead values, which are caused by executing the *performance simulation functions* as in Section 2. After the calculation, the results can be used to improve the main memory simulation algorithm. In the context of *dynamic performance stubs*, they are referred as *calibration functions*.

4.1 Measurement Tools

The test environment is based on a Linux operating system (see also Section 5). Hence, measurement tools, which can be used in most Linux systems, are described here. They are used to gather the information to determine the various overhead caused by the main memory simulation algorithm. In this section, the tools used for the *calibration functions* are presented shortly.

Allocated Heap Memory. To measure the behavior of the heap memory the “mallinfo” structure provided by the `malloc.h` header-file is used. This structure delivers several aspects about heap memory allocation such as the memory area, used and free amount of heap memory within the area. The mallinfo struct is read by calling the `mallinfo()`-function that is also part of the `malloc.h` header-file.

Allocated Stack Memory. The size of the allocated stack memory is measured by reading the base and stack pointer of the running process. Their difference represents the actual amount of stack allocated by the component under test. To read their values with the least overhead, some inline assembler code have been used.

Time. The measured values for allocated stack and heap memory have to be put in a chronological sequence. For that reason, the time has also

to be measured when picking heap and stack values. Within this paper, the real time clock (RTC) of the system is used.

4.2 Overhead Determination

This section presents a possibility to determine the three types of overhead, which are generated by the simulation algorithm: “time”, “heap” and “stack”.

4.2.1 Time Overhead

When simulating the memory behavior of the component under study the simulation time has to be almost same as the original runtime. But, there are several reasons why the execution time takes longer.

Reasons for Time Overhead. There are mainly five reasons for the time overhead produced by the algorithm:

- The execution time of the algorithm without any memory allocation
- The allocation of heap memory
- The use of the allocated heap memory
- The allocation of stack memory
- The use of the allocated stack memory

Running the algorithm consumes time, even if no memory is allocated. This basic time overhead is produced by the algorithm. When allocating heap memory the runtime of the algorithm will increase because of the time that is needed by the `realloc()`-function. Additionally, the use of the allocated heap memory by the `distmemset()`-function also takes some time. The same behavior can be regarded when the stack memory is allocated. The `alloca()`-function and the use of `distmemset()`-function produces additional time overhead.

Measurement. To measure the total time overhead ($time_{overhead}$) for the algorithm several independent measurements have to be done in order to cover all above mentioned cases. For each case, a simulation data file, to simulate the particular situation, is used.

Basic Execution Time. First of all, the evaluation of the basic execution time ($time_{basic}$) of the algorithm when no memory is allocated is performed. This is done by running the complete algorithm. To cover the complete algorithm a minimum stack allocation value of 1 byte is used. As

the heap should not be regarded within the $time_{basic}$ a value of 0 is applied. The time delay between two trace points is also set to 0 to measure the time consumption of the algorithm.

Measuring the basic time overhead for the algorithm shows a constant offset value for each run. This value is produced by running the algorithm’s code. Therefore, it has to be considered as the minimum time resolution that can be simulated by the algorithm. But, because of the other time overhead, e.g., produced by the allocation and use of the memory, this resolution can hardly be achieved.

Time Overhead Caused by Heap Allocation. The time overhead produced by the allocation of the heap memory ($time_{heap}$) is caused by the allocation itself and the use of the memory by the `distmemset()`-function. To evaluate this overhead, both parts of the overhead has to be measured separately. These measurements use an identical trace file to simulate heap allocation. The time delay is set to 0 and for stack allocation 1 byte is used at every trace point. This is done to run through the complete algorithm. The amount of allocated heap memory increases at each trace point.

To determine the time used for allocating and using the memory ($time_{heap}^{alloc}$ and $time_{heap}^{use}$) the time before and after the function call is measured as described above. Now, the “after” time is subtracted from the “before” time to evaluate the time delta.

Measuring the time overhead produced by the heap memory allocation highly depends on the architecture and the implementation of the used allocation function as described in [10]. For example, the change of the allocation options within the system influences the number of page faults produced and the way allocated memory is freed within the simulation.

Time Overhead Caused by Stack Allocation. To evaluate the time overhead that occurs when allocating stack memory ($time_{stack}$) the same measurements as shown within the heap allocation have to be done. There are two measurements, one to determine the time used to allocate the memory ($time_{stack}^{alloc}$) and another to measure the time to use the allocated memory ($time_{stack}^{use}$). The delay of the stack allocation is determined according to the heap allocation.

The only difference is the simulation data file that is used for the measurement. The time delay is set to 0 also in this *calibration function*. But, there should be no heap memory allocation. Hence, its value is set to 0, too. The amount of allocated stack memory is increasing throughout the simulation data file.

With these two measurements the timing behavior of the stack memory allocation can be determined. This again, strongly depends on hardware and implementation of the system.

Time Overhead in Total. In the previous paragraphs, a possibility to determine the various parts of the time overhead has been shown. These times have to be measured separately as different settings for the simulated memory and times have to be used. To get the total amount of time overhead, those parts have to be combined.

$$time_{overhead}(heap, stack) = time_{basic} + time_{heap}(heap) + time_{stack}(stack) \quad (1)$$

$$time_{heap}(heap) = time_{heap}^{alloc}(heap) + time_{heap}^{use}(heap) \quad (2)$$

$$time_{stack}(stack) = time_{stack}^{alloc}(stack) + time_{stack}^{use}(stack) \quad (3)$$

This is presented in Equations 1 - 3. ‘‘Heap’’ and ‘‘stack’’ denotes the amount of bytes, which will be allocated/used.

4.2.2 Byte Overhead

The execution of the simulation algorithm uses a certain amount of memory. Therefore, the overhead for allocated heap and stack memory have to be measured.

Heap Overhead. To measure the heap memory overhead ($heap_{overhead}$), created by the running algorithm, a specific simulation data file with increasing heap memory allocation is used. The time delay is not relevant here and thus set to zero. The amount of allocated stack memory is set to 1 byte in order to execute the complete algorithm.

The measurement results show, that using the algorithm as described in Section 2 creates only a small overhead in allocated heap memory.

Stack Overhead. As done for the heap memory, the overhead of the allocated stack memory has to be determined, too.

The overhead regarding stack memory that occurs when executing the algorithm is mainly caused by two reasons. On the one hand, there are local variables used to initially set up the simulation. They are located on the stack and thus produce a certain amount of allocated stack memory. This is called ($stack_{offset}$). On the other hand, every function call of the allocate()-function increases the allocated stack memory. This is called ($stack_{overhead}$). This is an important aspect, due to the recursive simulation algorithm, which produces several function calls depending on the simulation data points.

Measuring the stack memory overhead, both $stack_{offset}$ and $stack_{overhead}$, can be done by using a trace file that increases the amount of allocated stack continuously. Time and heap values are not considered within this measurement. To estimate the influence of the recursive function calls, the stack memory is allocated several times in a row before freeing it again.

The different measurements show that the amount of additional stack memory allocated by the algorithm (*stack_{overhead}*) is constant for every call of the `allocate()`-function and, therefore, for its recursive call as well.

5 Proof of Concept

In the previous sections an algorithm to simulate a program’s memory behavior has been presented and the *calibration functions* have been introduced. Now, both will be validated and evaluated within this proof of concept. Therefore, a defined sample for the input data is used to cover a broad variety of possible memory behaviors.

5.1 Experimental Setup

This proof of concept is used evaluate to the impact of the *calibration functions* as well as the enhanced main memory allocation algorithm.

Environment. All measurements were performed on a FSC Amilo Si3655 Notebook with an Intel Core(TM)2 Duo P8400 CPU (Intel 64 architecture). As operating system Arch Linux is used. Its kernel version is 2.6.34. The binary has been build using the gnu compiler collection (gcc) without any optimization flags to guarantee that the option “-O0” has been used. Beside of running the proof of concept, the system has been idle to avoid further influences on the execution time.

Measurement Tools. To offer the possibility to evaluate the simulated behavior of the memory allocation a very precise way to measure the stack and heap allocation has to be used.

For this reason, the value of allocated stack memory is measured by inline assembler calls to read the stack pointer (esp) and base pointer (ebp) registers. The value of ebp is taken at the beginning of the simulation to get a base value for the stack allocation. During the simulation the esp register has been read at every measuring point. So the offset between the starting ebp and the actual esp gives the actual total amount of allocated stack memory.

To measure the value of allocated heap memory, the mallinfo structure of the malloc.h header-file is read. This structure contains all the desired information about the heap memory for this process.

The measured data has to be associated with the time spent in the system. Because of this, at every measuring point a time stamp is taken using an inline assembler to read the real time clock of the system [11].

5.2 Calibration Function

To determine the time, heap and stack allocation offset, created by executing the simulation, the *calibration functions* as presented in Section 4 are used. The values for those offset depends on the system's implementation. Hence, the calibration has to be repeated when changing any of the system's parameters. As described in Section 4, different simulation data files have to be used to measure the various offset values of the algorithm.

Time Overhead. The measurement of the basic time offset has shown to be constant in our setup. It is determined to $time_{basic} = 126775$ cycles with an squared coefficient of variation (see also [12]) of 0.009, calculated for 100 test evaluations.

Within this proof of concept, the time consumed by allocating stack and heap memory has been identified. The evaluation of the measurements, that were described in Section 4, leads to following results for the heap and stack allocation (y describes the previously allocated total memory size in the memory segment and x the newly allocated memory in bytes).

Equation 4 is used to calculate the number of page faults at a certain memory allocation value.

$$pagefaults(x, y) = \left\lfloor \frac{(y \% pagesize) + x}{pagesize} \right\rfloor \quad (4)$$

The number of bytes, which did not cause a page fault is calculated ($y \% pagesize$). The result plus the newly allocated memory (x) is divided by the $pagesize$ to determine the amount of page faults for the new allocation. The result is passed to the floor function as page faults can only be a natural number. $Pagesize$ denotes the system page size in bytes.

Time Influence of the Heap Simulation. The heap memory will only be reallocated. Hence, the memory value (x) is always greater or equal zero.

As can be seen in Equation 5, the time spent for allocating memory heavily depends on whether a page fault is raised in the allocation function or not. Additionally, there is only one page fault in the allocation function even if more than one page is allocated.

$$time_{heap}^{alloc}(x, y) = \begin{cases} 3722cycles & pagefaults(x, y) > 0 \\ 94cycles & pagefaults(x, y) = 0 \end{cases} \quad (5)$$

$$time_{heap}^{use}(x, y) = 69cycles * (pagefaults(x, y) + 1) + 3252cycles * \begin{cases} pagefaults(x, y) - 1 & pagefaults(x, y) > 1 \\ 0 & pagefaults(x, y) = 0 \end{cases} \quad (6)$$

In Equation 6, the time spent in the `distmemset()`-function is calculated. The equation consists of two parts. First, the time spent iterating over the memory block, i.e., $69\text{cycles} * (\text{pagefaults}(x, y) + 1)$ and, second, the number of page faults occurred in the function minus one as one page fault appeared within the allocation function (see also Equation 5).

Time Influence of the Stack Simulation. For both times, i.e., $\text{time}_{stack}^{alloc}(x)$ and $\text{time}_{stack}^{use}(x)$, the algorithm does not take significant time to free the stack memory ($x \leq 0$). Additionally, the “freed” memory will not be used, obviously. Hence, both values are set to zero cycles. In the other case, the time needed to allocate and use the new memory can be calculated by using Equations 7 and 8.

$$\text{time}_{stack}^{alloc}(x) = 48\text{cycles} \quad (7)$$

$$\begin{aligned} \text{time}_{stack}^{use}(x, y) = & 61\text{cycles} + \\ & 3358\text{cycles} * \text{pagefaults}(x, y) \end{aligned} \quad (8)$$

The time to allocate stack memory (Equation 7) is constant as only the base- and stack pointer have to be adjusted [9].

The time spent in the `distmemset()`-function to initialize the stack memory (Equation 8) is the same as in Equation 6. The only difference is that the stack allocate function does not raise a page fault.

All the described equations were found by determining the average time stamps of several runs in our test setup and describe the time behavior of allocating and using the heap and stack memory in sufficient accuracy.

When not allocating any heap and/or stack memory at a trace point, the respective times are set to 0. In those cases, they do not have any influence on the calculation of the total time overhead for each measuring point. The equation used to determine the total time overhead $\text{time}_{overhead}(\text{heap}, \text{stack})$ is described in Section 4.

Heap and Stack Overhead. As stated in Section 4, the heap offset that is introduced when executing the algorithm has to be determined. The measurements showed that $\text{heap}_{overhead}$ is constant at 32 bytes, if there is no heap memory allocated within the simulation. If there is any heap memory allocated during the simulation, the heap overhead rises to 40 bytes and also remains constant while the memory is allocated.

Measuring with the given calibration trace file results in a constant increase of allocated stack per trace point. So, the call of the allocation function allocates a constant amount of stack memory. Because of this measurement, the stack offset is determined to $\text{stack}_{offset} = 216$ bytes as well as to

$$stack_{overhead}(x) = \begin{cases} 64\text{bytes} & x > 0 \\ 0\text{bytes} & x = 0 \\ -64\text{bytes} & x < 0 \end{cases}.$$

As these bytes overhead are constant for each execution, there is no need for an statistical interpretation.

The values for stack, heap and time overhead is used to produce a simulation data file. This allows the simulation algorithm to perform a simulation that fit as exact as possible to the desired behavior of memory allocation.

5.3 Measurement and Evaluation

After the measurements of the calibration functions, all necessary data for the simulation of the memory behavior is available. The same input data as in [9] has been used and the time, heap and stack overhead has been determined via the *calibration functions*.

Simulation Data File. As the first step in simulating the memory behavior of a system, a valid simulation data file has to be generated. With the input data and the overhead for time delay, heap and stack allocation, the needed trace points are calculated. The output is presented as a header file, see Section 2.3, containing the data set used within the simulation algorithm.

Measurement. After creating a valid header file, an executable of the simulation algorithm can be build.

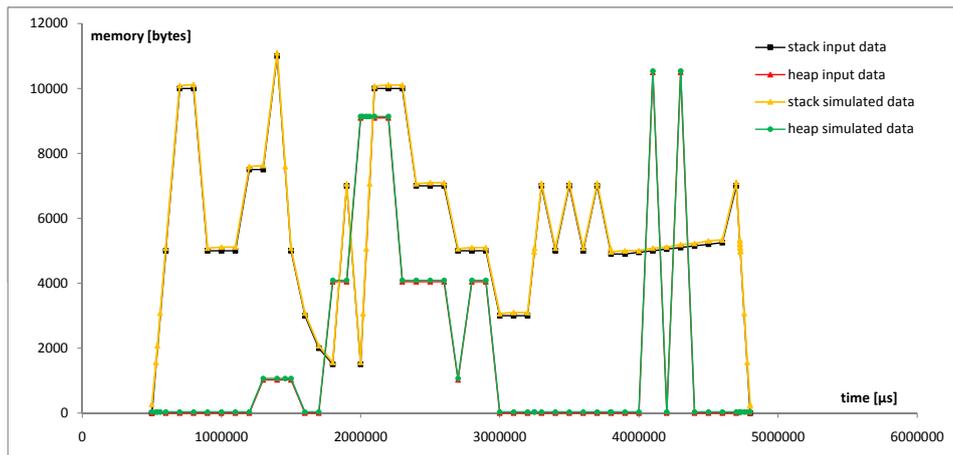


Figure 3: Comparison of Original and Simulated Memory Behavior

Figure 3 shows the measured stack and heap memory allocation in comparison to the desired behavior. The time in microseconds is printed at the X-axis and the total allocated amount of memory is shown at the Y-axis. The figure shows the original stack behavior, the measured stack allocation during the simulation, the original heap allocation and the measured heap behavior while simulating the memory allocation.

Simulating Execution Time. When comparing the time supposed by the simulation data file, which is 4.8 seconds, and the execution time measured in the evaluation, which is 4.8000444 seconds, it can be seen that the simulation produces only a small amount of time overhead. Here, an overhead of $9 * 10^{-3}\%$ in total execution time is produced. So, the total execution time is sufficiently simulated.

Simulating Heap Allocation. The analyses of the heap's allocation simulation, as shown in Figure 3, depicts that it is very accurate. There is nearly no variation to the desired behavior of heap allocation. It is possible to simulate situations where the heap is rising and falling. Fast switches of allocating and freeing heap memory are simulated exactly. The simulation algorithm works absolutely fine for simulating heap memory allocation in our example.

Simulating Stack Allocation. The results of the simulation of stack memory allocation behavior also are satisfying. The allocation behavior can be reproduced exactly. Rising and trailing edges as well as constant amounts of stack are simulated in a correct way. High peaks and fast changes of allocated stack are rebuild as desired. Even slow rises of the allocated amount of stack are simulated quite well.

Summary. The *calibration functions* that are introduced within this paper as well as the presented memory simulation algorithm fully meet the requirements to simulate the memory behavior of a system under test. Both, heap and stack memory allocation, are simulated with high accuracy and almost without an overhead in execution time.

6 Conclusion and Future Work

This paper presents an algorithm to simulate the main memory behavior of applications in the context of *dynamic performance stubs*. There are two contributions: An extension and improvement of the simulation algorithm and the determination of the various types of overhead caused by the algorithm, e.g., time overhead caused by executing the algorithm.

With the improvement of the algorithm, it is now possible to simulate constant stack behavior. Additionally, *calibration functions* have been intro-

duced to evaluate the overhead values caused by executing the algorithm. Considering the *calibration functions* leads to an almost exact simulation of the main memory behavior. This has been validated with a proof of concept.

The future work will focus on an algorithm to measure the component under study as well as to generate the simulation data file. Moreover, a methodology, which applies the *main memory stubs* in industrial case studies, has to be defined and evaluated.

It has been shown that the behavior of the stack and heap usage can be simulated without significant errors. Based on the presented algorithm, a goal oriented performance optimization regarding the memory behavior of an arbitrary application can be achieved.

7 Acknowledgments

The authors would like to thank the long term evolution 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 Helge Janicke. 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

- [1] P. Trapp and C. Facchi, “Performance Improvement Using Dynamic Performance Stubs,” Fachhochschule Ingolstadt, Tech. Rep. 14, Aug. 2007.
- [2] R. Jain, *The art of computer systems performance analysis*. Wiley and sons, Inc., 1991.
- [3] N. H. Gunther, *The Practical Performance Analyst*. McGraw-Hill Education, 1998.
- [4] J. J. Marciniak, *Encyclopedia of Software Engineering*, 2nd ed. John Wiley & Sons Inc, 2002.
- [5] 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.

- [6] I. Sommerville, *Software Engineering*, 6th ed. Addison-Wesley, 2001, german redaction.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] P. Ezolt, “A study in malloc: a case of excessive minor faults,” in *ALS '01: Proceedings of the 5th annual Linux Showcase & Conference*. Berkeley, CA, USA: USENIX Association, 2001, pp. 17–17.
- [11] Y. Etsion and D. Feitelson, “Time Stamp Counters Library - Measurements with Nano Seconds Resolution,” The Hebrew University of Jerusalem, Tech. Rep. 2000-36, 2000.
- [12] 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.

Biography



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



Dipl.-Inf. (FH) Markus Meyer is a research assistant at the Institute for Applied Research of the UAS Ingolstadt. His research interests are in the area of performance and software engineering. Markus Meyer was born in 1983 at Eichstätt, Germany. He has a diploma degree in Computer Science from the UAS Ingolstadt and worked as a software engineer before he changed to the UAS Ingolstadt.



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

Impressum

Herausgeber

Der Präsident der
Hochschule für angewandte
Wissenschaften FH Ingolstadt
Esplanade 10
85049 Ingolstadt
Telefon: 0841 9348-0
Fax: 0841 9348-200
E-Mail: info@haw-ingolstadt.de

Druck

Hausdruck

Die Beiträge aus der Reihe „Arbeitsberichte – Working Papers“ erscheinen in unregelmäßigen Abständen. Alle Rechte, insbesondere das Recht der Vervielfältigung und Verbreitung sowie der Übersetzung vorbehalten. Nachdruck, auch auszugsweise, ist gegen Quellenangabe gestattet, Belegexemplar erbeten.

Internet

Dieses Thema können Sie, ebenso wie die früheren Veröffentlichungen aus der Reihe „Arbeitsberichte – Working Papers“, unter der Adresse www.haw-ingolstadt.de nachlesen.

ISSN 1612-6483