Performance Simulation of a System's Parallelization

Markus Meyer*, Helge Janicke†, Peter Trapp*, Christian Facchi* and Marcel Busch*
*Institute of Applied Research, University of Applied Sciences Ingolstadt, Germany
{markus.meyer, peter.trapp, christian.facchi, marcel.busch.winf}@haw-ingolstadt.de
†Software Technology Research Laboratory, De Montfort University, Leicester, United Kingdom
heljanic@dmu.ac.uk

***Abstract*—A new approach to simulate the parallelization of a software function is presented in this paper. The parallelization's effects onto the system's performance prior to a costly realization of the parallelization are evaluated, leading to a more gain-oriented performance optimization. The presented approach defines a methodology to transform a single-threaded software function into a multi-threaded simulation. *CPU stubs*, simulating both, the performance and functional behavior, are applied to simulate the expected time slices. The proposed technique can estimate the expected performance gain for the whole system. A proof of concept is used to evaluate the proposed methodology and the simulation results are compared to a known parallel implementation of the algorithm. Initial results show our approach can be used to simulate the performance behavior of a parallelized system with high accuracy. In addition, the number of threads that result in the highest performance gain of the system is determined.**

Keywords-software performance optimization; performance simulation; parallelization; dynamic performance stubs

I. Introduction

Dynamic performance stubs [1] can simulate various optimization levels of the component under study (CUS). *CPU stubs* [2], replace a software function (CUS) in the system under test (SUT) and model its CPU performance behavior. *CPU stubs* consist of two functionalities, the *simulated software functionality* (SSF), that recreates the functional behavior of the bottleneck, and the *performance simulation functions* (PSF) that simulate its performance behavior [3]. *PSF* are used to simulate different optimization levels and show the optimization's effect on the system's performance.

This paper presents a novel approach to the analysis of an identified bottleneck with respect to its parallelization potential. The key contribution of this work is that it allows for informed decision making to what degree the optimization of the bottleneck using parallelization techniques would impact on the overall performance of the system.

Section II shows the related work. The presented approach (Section III) is based on a simulation of the bottleneck. It is an efficient technique that can help to avoid wasted effort and costs associated with the parallelization of the affected system component. The described methodology is evaluated by a proof of concept in Section IV. The simulation of a system's parallelization can be used to estimate the achievable performance using parallelization techniques.

II. Related Work

The concept of *dynamic performance stubs* was introduced in [1]. In [3], the usage of *CPU stubs* to determine the performance gain of a system's optimization is shown. Our approach, extends the *dynamic performance stubs* to investigate if this performance gain can be achieved by parallelization. According to Amdahl [4], not all instructions that are executed within a system can be performed concurrently. Amdahl shows that the maximum speedup of a parallelization is limited by the sequential part of an algorithm for an infinite number of processors. Gustafson [5] claims that the speedup that can be obtained by a parallel execution is not only limited by the remaining sequential part but increases linearly by the number of used processors. Both, [4] and [5], determine an upper bound of the speedup achievable with parallelization. In contrast, our approach simulates the anticipated system-wide performance benefit. In our case, the speedup described by Amdahl and Gustafson can not be used to estimate the runtime of the parallel parts of the algorithm without modification (see [6]).

The parallelization of a sequential algorithm requires thread management that reduces the achievable speedup [7]. Marinescu and Rice [8] introduced the concept of relative speedup taking sequential and duplicated work, communication and control and blocking into account. This paper's approach also considers the overhead created by parallelization. By simulating the parallelization, system influences such as the number of available processors and the current system load are included.

In [9], an approach to model the influences of the number of threads and processors on a system's performance using Solaris containers is introduced. In contrast, the presented approach simulates the parallelized software within the real system in order to get accurate performance measurements. To the best of our knowledge, no studies have evaluated the overall performance gain that can be achieved within the system using parallelization techniques.

III. Simulating parallelization

This section presents an approach to simulate the parallelization of the system in order to obtain measurements on which decisions to use parallelization as an optimization

 ISBN: 978-1-61208-165-6

technique can be well-founded. Based on the results the degree of parallelization can be determined.

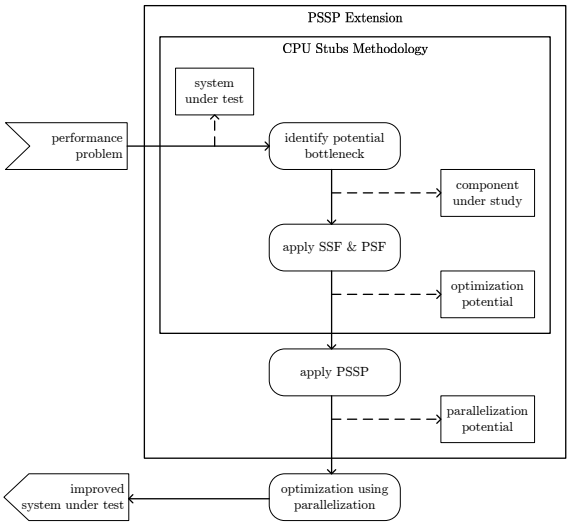

Figure 1. "Performance Simulation of a System's Parallelization" (PSSP) - Extension of the CPU Stubs' Methodology

Figure 1 depicts our approach. The *performance simulation of a system's parallelization* (PSSP) extends the methodology of *CPU Stubs* [2]. *CPU Stubs* determine the optimization potential of a system that is affected by performance issues. A potential bottleneck (component under study, CUS) is identified within the system under test (SUT). The CUS is replaced by a *CPU Stub* simulating the functional and the performance behavior of the CUS. By varying the stub's performance behavior, the *optimization potential* for this component is determined. The approach presented in this section extends the known methodology by the simulation of the system's parallelization to decide whether the *parallelization potential* within the CUS is sufficient to reach the performance optimization goals. Hence, the presented approach helps to transform the measured overall optimization potential into a gain-oriented realization of the performance optimization.

A. Objectives

Depending on the accuracy of the known and measured original performance data, the simulation of the parallelization pursues the following objectives:

- **Performance potential:** Investigate if it is possible to gain performance within the system when parallelizing the CUS.
- **Determine the expectable performance gain:** Determine the performance gain that can be achieved by parallelizing the CUS. The system load and the available hardware resources limit the possible performance of the system. Decide whether the performance targets can be reached by a parallelization.
- **Determine the maximum overhead:** Parallelizing a sequential algorithm results in additional effort for thread management. The overhead needed to manage the parallelization reduces the achievable speedup.
- **Determine the optimal number of threads:** The number of used threads influences the overhead needed to manage the parallelization. The optimal number of threads that entails a short thread runtime in combination with a small amount of overhead is determined by test series.

B. Approach

The presented approach is based on the simulation of a concurrent implementation of the system component containing the bottleneck (CUS). It uses a simple fork and join mechanism to give a first estimation if parallelization is a valuable way to optimize the CUS. The introduced method applies the parallelization of the system using domain decomposition. Several threads are created within the simulation using the same thread runtime.

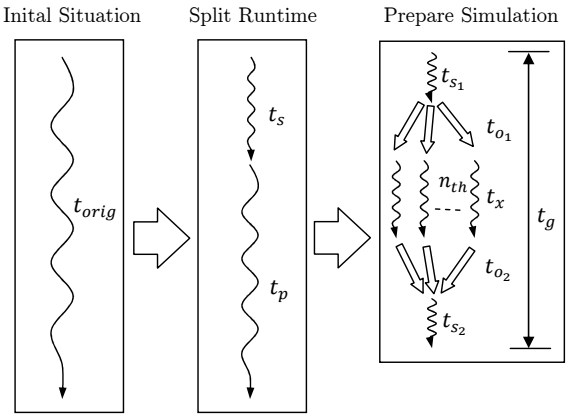

Figure 2. Steps Towards the Parallel Simulation

In Figure 2, the sequential time is broken down and replaced by a time estimation of the parallel computation.

Initial Situation A sequential implementation of the CUS is assumed. Its runtime t_{orig} is determined by measurements, e.g., using real time counters (see [10]).

Step 1: Algorithm Parallelization Potential The original runtime (t_{orig}) is split into a sequential part (t_s) and a parallel part (t_p) (see [4]), by carrying out data and control flow analysis (see [11], [12]).

In the next steps, the parts of the bottleneck that can be parallelized (t_p) as well as the sequential part (t_s) are prepared for multi-threaded execution. For that, the distribution of the sequential part, the threads' runtime (t_x) and the additional work that is needed to manage the threads (t_o) are determined.

Step 2: Sequential Part The control flow analysis, applied in the first step of this approach, can be used to determine the distribution of the sequential part to the

beginning and the end of the simulation. The total sequential time remains constant at $t_s = t_{s_1} + t_{s_2}$.

Step 3: Threads' Runtime The parallel time (t_p) is converted into the threads' runtime (t_x) considering that:

- each thread has to execute a subset of the original instructions. Therefore, the determined parallel execution time (t_p) will also be split to the various threads. This has to be captured by the parallel simulation of the algorithm. The threads' runtime (t_x) is a function of the number of threads: $t_x(n)$ where $0 \leq n \leq n_{th}$.
- the algorithm in the bottleneck rarely has linear time complexity. The problem size remains constant for the test case and is equally split in the n_{th} threads. However, the threads' runtime does not change in a linear way for algorithms with non-linear time complexity when changing the problem size. Due to that, the threads' runtime not only depends on the number of created threads (n_{th}), but also on the parallelized algorithm's complexity.

There are two approaches to determine the threads' runtime. A complexity analysis of the CUS and the measurement of the algorithm's time behavior for various problem sizes can be performed:

Complexity Analysis: A complexity analysis of the CUS's algorithm is applied to determine the time complexity of the algorithm using Big-O-Notation (see [13]). Using the complexity, the threads' runtime t_x can be calculated depending on the number of threads. The following example assumes that an algorithm with a time complexity of $O(n^2)$ is used and that $n_{th} = 2$ threads are created within the simulation.

$$t_p \leq c * n^2 \tag{1}$$

Equation 1 shows the formula for the parallel execution time (t_p). The time complexity given in the Big-O-Notation defines an upper bound for the time. Therefore, the parallel execution time is less or equal to a constant time (c) multiplied by the given complexity.

This paper's approach uses domain decomposition to simulate the system's parallelization. This is realized by a divide-and-conquer strategy (see [13]). Thus, the presented approach uses the same algorithm but with an reduced problem size per thread to simulate the parallelization. As shown in Equation 2, the reduced problem size is used to calculate the execution time.

$$t_x \leq c * (\frac{n}{n_{th}})^2 \leq c * (\frac{n}{2})^2 \tag{2}$$

The combination of Equations 1 and 2 shows that in this example the threads' runtime (t_x) only depends on the parallel runtime (t_p). The upper bound $t_p = c * n^2$ of the parallel runtime is used as a pessimistic estimate to calculate the threads' runtime in Equation 3.

$$\frac{t_x}{t_p} \leq \frac{c * (\frac{n}{2}^2)}{t_p} \leq \frac{c * n^2}{4 * t_p} \leq \frac{t_p}{4 * t_p} \leq \frac{1}{4} \tag{3}$$

The upper bound of the calculated thread runtime is $t_x = \frac{t_p}{4}$.

Measured Time Behavior: Another approach to determine the threads' runtime is the measurement of the algorithm's time behavior. The CUS's algorithm is available and can be executed with different problem sizes capturing the various time behaviors of the algorithm. This data is used to determine the threads' runtime for a given problem size. The measurements' results strongly depend on the used input data, as best- or worst-case scenarios of the algorithms can be triggered. To get the expected timing data, the input data has to be chosen specifically corresponding to the original data.

Step 4: Parallelization's Overhead In addition to the threads' runtime (t_x) the time used to fork and join the threads as well as to split and combine the result of the problem has to be considered within the simulation as overhead $t_o = t_{o_1} + t_{o_2}$. The overhead is split into two sections: creation (t_{o_1}) and synchronization (t_{o_2}) of the multi-threaded part (Figure 2). The overhead depends on the concrete implementation. Two options need to be considered:

1) An idea about the implementation of the parallelization is available. Hence, the effort that is needed for the thread management can be estimated depending on the number of created threads.
2) Otherwise, it is possible to simulate the parallelization without an overhead. In this case, only the maximum overhead that can be introduced by the parallelization in order to gain performance can be determined. This reduces the problem to a thread-management and task distribution problem that can be solved more easily.

Depending on these considerations, the evaluation of the results focuses on different objectives (see Step 6).

Step 5: Simulation of the Parallelization The determined values of t_s, t_o and t_x are used to simulate the performance of a possible parallelization (Figure 2). Each of these elements has to be rebuild as a *system influencing PSF* [2]. To build the multi-threaded behavior of the simulation, a method to generate the *multi-threaded PSF* from a textual description is introduced in [3]. In combination with the *SSF* [3], it is possible to simulate the expected performance behavior of a parallelization. The expected parallel execution time is $t_g = t_s + t_o + t_x$. This theoretical value does not consider any scheduling or runtime influences. The time t_g^{sim}, measured within the simulation under real conditions describes the real value of t_g. Consequently, the simulation takes system bottlenecks, as the number of available processors or other processes running on the system, into account.

Step 6: Evaluation of the Results Depending on the overhead estimation (Step 4) two results of the simulation can be evaluated:

- The performance gain by parallelization of the bottleneck is defined as $t_{gain} = t_{orig} - t_g^{sim}$. The simulation

is executed with a varying number of threads (n_{th}) to determine the optimal gain, leading to adjusted values for t_x and t_o. This is used to determine the optimal number of threads (n_{th}^{opt}).

- If the threads' administration overhead cannot be estimated (Step 4.2) the determination of maximum overhead is another objective. In this case the simulation rebuilds the sequential and parallel execution times, t_s and t_x and estimates the maximum overhead by $t_o^{max} = t_{orig} - t_g^{sim}$. If the parallelization and synchronization of the bottleneck can be performed in less than t_o^{max}, the optimization will result in a performance improvement for the bottleneck.

These results are used to decide whether parallelization can be applied as an optimization technique to achieve the performance targets.

IV. Proof of Concept

This section evaluates the proposed methodology using a proof of concept. The threads' runtime is determined as described in Step 3. The measurements are used to simulate the parallelization for various numbers of threads. Finally, the simulation's results are compared to the performance data of the algorithm's parallel implementation. 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, kernel version 2.6.34, is used. Binaries were build using GCC (version 4.5.1).

A. Overview

The example simulates the parallel execution of a sequential bubble sort algorithm (see [14]). A parallel implementation of this algorithm is used to validate the results of the simulation and uses the same implementation extended by a merge-sort to combine sorted sublists.

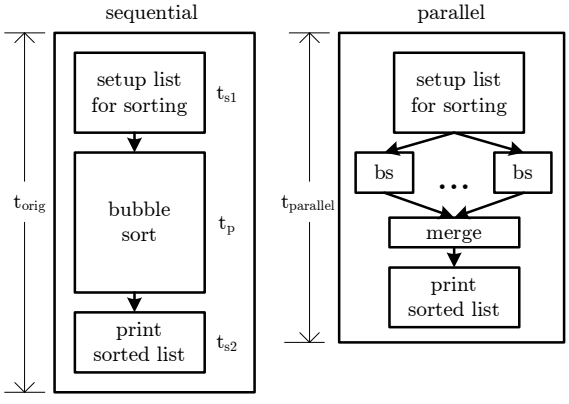

Figure 3. Sequential and Parallel Bubble Sort Algorithms

Figure 3 shows the timings of the bubble sort algorithms. Both algorithms have the same set up. The sequential algorithm processes the whole list, whereas in the parallel case, the list is split into n_{th} parts that are sorted independently in different threads. The parallel execution additionally merges the sorted partial lists. Finally, both algorithms print the whole sorted list. To guarantee reproducibility of the test results, the used list is initialized with 4096 random integer numbers once and used for each test run.

B. Algorithm Parallelization Potential (Step 1)

The original runtime (t_{orig}) of the sequential algorithm was determined using the time stamp counters. Additionally, the runtime of the parallel part (t_p) and the sequential parts of the algorithm $t_s = t_{s1} + t_{s2}$ were measured as described in Step 1.

	avg[ms]	max[ms]	min[ms]	sqdcoeff of var
t_{orig}	103,66	103,87	103,48	4,06E-07
t_p	103,49	103,70	103,31	4,04E-07
t_{s1}	0,152	0,155	0,150	9,87E-05
t_{s2}	0,0177	0,0177	0,0175	1,63E-05

Table I
Runtime Values of the Sequential Bubble Sort

Table I shows the measurements' average, maximum and minimum time of 20 samples taken in milliseconds. The squared coefficient of variation (see [15]) is applied to evaluate the deterministic behavior of the measured data.

C. Sequential Part (Step 2)

The sequential time t_s was measured (Section IV-B). Thus, t_{s1} and t_{s2} are known and can be simulated.

D. Threads' Runtime t_x (Step 3)

In the next step, the measured parallelizable time of the sequential algorithm (t_p) has to be converted into the threads' runtime (t_x). As described in Step 3 of Section III-B, there are two possibilities to estimate t_x, *complexity analysis* and *measurement*. To evaluate that both approaches can be used to determine the threads' runtime, their results are compared in this step. The complexity of the bubble sort algorithm is known as $O(n^2)$ [14]. In Section III-B, the calculation of the estimated thread runtime is shown by an example. To measure the time behavior of the used bubble sort algorithm, it is executed with different problem sizes.

Figure 4 presents the comparison of the determined parallel threads' runtime t_x. As described in Step 3 of Section III-B, the calculation of the thread's runtime using a complexity analysis (crosses) and its determination by measurements (circles) are drawn. The x-axis shows the number of threads, whereas, the y-axis depicts the time in milliseconds. In order to depict the differences of the two graphs, the y-axis is drawn logarithmically. As can be seen, both alternatives nearly provide the same results and can be used to give an estimation about the threads' runtime. As described in Step 3 of Section III-B, the complexity analysis provides an upper bound for the thread's runtime.

 ISBN: 978-1-61208-165-6

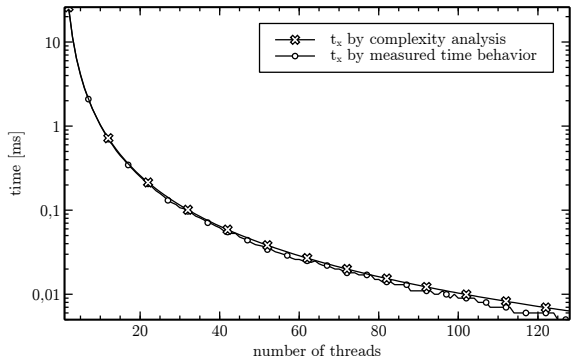

Figure 4. Determination of the Threads' Runtime

E. Parallelization's Overhead (Step 4)

The overhead that is needed for thread management has to be determined in this step. In this example, the parallel implementation of the algorithm is known. Due to that, the overhead values t_{o1} and t_{o2} can be estimated. The overhead to split the initialized list to the single threads is measured and set to $t_{o1} = 5,2\mu s$, as the additional work that has to be done here is almost constant. The single bubble sort threads are just executed with sublists.

The work needed to synchronize the intermediate results influences the expected performance gain. As described in Section IV-A, a merge algorithm with a complexity of $O(n_{th} * n)$ is used to combine the results. In this example the overall problem size (n) remains constant. Thus, the time to merge the sublists only depends on the number of created threads. This overhead increases linearly and can be determined using the same methods as described for the calculation of the threads' runtime in Step 3 of Section III-B. In this proof of concept, it was determined to an average of $t_{o2} = 28\mu s * n_{th}$.

F. Simulation and Evaluation (Steps 5 & 6)

After all values have been determined, the simulation was executed. All measurements were performed with the number of threads increasing from 2 to 128.

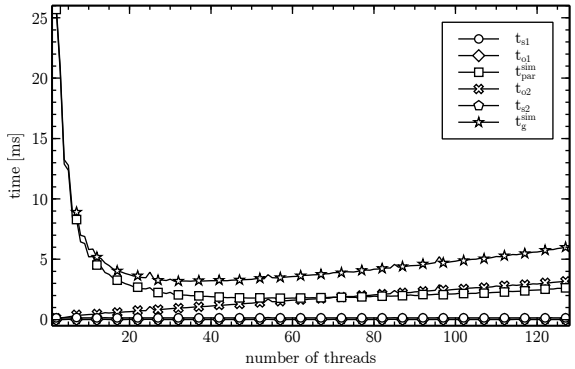

Figure 5. Simulation of the System's Parallelization

Figure 5 presents the simulation's results. The x-axis shows the number of created threads and the y-axis depicts the used time in milliseconds. The graph includes all the values introduced in Section III-B; the sequential parts t_{s1} and t_{s2}, the threads' overhead t_{o1} and t_{o2}, as well as the time needed to run all the created threads t_{par}^{sim}. Additionally, the overall simulation time (t_g^{sim}) is presented. The small variations included in the graph occur due to measuring inaccuracies.

The evaluation of the simulation resulted in a calculated performance gain of $t_{gain} = 100.48ms$ for the parallel execution with the optimal number of threads $n_{th}^{opt} = 35$.

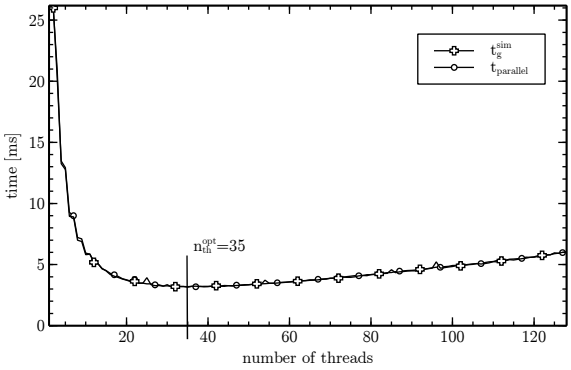

Figure 6. Comparison of Simulation and Parallel Implementation

Figure 6 presents a comparison of the overall simulation time t_g^{sim} and the total execution time $t_{parallel}$ of the parallel implementation. The y-axis plots the time in milliseconds against the number of used threads (x-axis). There are only small differences, probably caused by operating system interruptions. Clearly, the simulation of the parallelization has given an accurate estimation of the parallelization's results. Based on these initial results we are confident that the proposed methodology can be used to estimate the expected performance of a system's parallelization. The application of the presented approach provides a well founded estimation of the ideal number of threads without a realization of the parallel algorithm.

V. Conclusion and Future Work

We presented a novel approach to simulate the effects of parallelization on the system's performance, by obtaining measurements on which decisions to use parallelization as an optimization technique can be based. This leads to a more gain-oriented performance optimization.

The presented approach converts a single-threaded bottleneck into a simulation of its parallel execution. Therefore, the expected bottleneck is split into several parts in order to simulate the parallel execution of the bottleneck. Especially, the part that can be parallelized is particularly investigated to determine the threads' runtime and the additional overhead that occurs when synchronizing the threads' results.

 ISBN: 978-1-61208-165-6

Complexity analysis and measurements are used to estimate the timing behavior of the parallel part. The identified time slices are rebuild by *performance simulation functions*, and, thus, enable the performance simulation of the system's parallelization. The parallelization's expected performance benefit is determined by simulating varying parallelization degrees.

The described simulation provides more accurate knowledge on the achievable performance gain by parallelizing a component before spending the effort of a costly realization of the parallelization. And such, it enables a more gain-oriented performance optimization than a simple guess of the parallelization's effects. The approach is validated by a proof of concept using a bubble sort algorithm. The simulation's results estimate the expected performance of the parallel implementation with high accuracy. Additionally, the optimum number of threads that have to be created in order to achieve the maximum performance benefit is determined by the simulation. Please note that in a real environment a comparison of the expected performance gain and the parallel runtime behavior, as shown in the presented case study, is not possible until the realization of the parallelization.

In future work the introduced approach also has to be validated for other algorithms and in industrial case studies. Additionally, the CPU load will be taken into account as a further performance aspect. Upper bounds of CPU loads can be defined and help to decide whether parallelization of a system's component is a viable approach for the system's performance optimization.

The presented approach simulates a parallelization of the regarded bottleneck using domain decomposition. Functional decomposition as a parallelization technique will also be investigated, considering the simulation of threads with differing values of runtime and systems having several synchronization points of the threads during their execution. This has to be integrated with the presented approach.

A new methodology regarding the application of the performance simulation of a system's parallelization will be developed and evaluated by case studies. This enables the simulation of parallelization as a performance optimization technique. Thus, it reduces the optimization's effort and leads to a more gain-oriented performance optimization of the system.

VI. Acknowledgments

The authors would like to thank the Software Technology Research Laboratory (STRL) of the De Montfort University, especially Francois Siewe and Hussein Zedan for providing the appropriate environment for research. This research has been funded by project grants from the German Federal Ministry of Education and Research (BMBF). Project: PerfBoost, FKZ 17034X10.

References

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

[2] P. Trapp, M. Meyer, and C. Facchi, "Using CPU Stubs to Optimize Parallel Processing Tasks: An Application of Dynamic Performance Stubs," in *ICSEA '10*. IEEE Computer Society, 2010.

[3] P. Trapp, M. Meyer, C. Facchi, H. Janicke, and F. Siewe, "Building CPU Stubs to Optimize CPU Bound Systems: An Application of Dynamic Performance Stubs," *International Journal on Advances in Software*, vol. 4, no. 1&2, 2011, accepted for publication.

[4] G. M. Amdahl, "Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities," in *AFIPS '67 (Spring)*. New York, NY, USA: ACM, 1967, pp. 483–485.

[5] J. L. Gustafson, "Reevaluating Amdahl's Law," *Commun. ACM*, vol. 31, pp. 532–533, May 1988.

[6] S. Krishnaprasad, "Uses and Abuses of Amdahl's Law," *J. Comput. Small Coll.*, vol. 17, pp. 288–293, December 2001.

[7] C. Jeong and M. Shahsavari, "Performance Evaluation of Multithreading in Concurrent Programs," in *SoutheastCon, 2002. Proceedings IEEE*, 2002, pp. 7 –9.

[8] D. Marinescu and J. Rice, "Speedup, Communication Complexity and Blocking-a La Recherche du Temps Perdu," in *Parallel Processing Symposium, 1993., Proceedings of Seventh International*, apr 1993, pp. 712 –721.

[9] A. Siami Namin, M. Sridharan, and P. Tomar, "Predicting Multi-core Performance: a Case Study Using Solaris Containers," in *Proceedings of the 3rd International Workshop on Multicore Software Engineering*, ser. IWMSE '10. New York, NY, USA: ACM, 2010, pp. 18–25.

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

[11] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, 2006.

[12] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1999.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[14] O. Astrachan, "Bubble Sort: an Archaeological Algorithmic Analysis," in *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, ser. SIGCSE '03. New York, NY, USA: ACM, 2003, pp. 1–5.

[15] 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.

 ISBN: 978-1-61208-165-6