# Research Proposal: Performance Engineering


| | |
|---|---|
| **Institution:** | Department of Computer Science, University of Copenhagen (DIKU) |
| **Proposed duration:** | 1.1.1999 – 31.12.2001 |
| **Principal investigator:** | Jyrki Katajainen, Assoc. Prof. |
| **Other investigators:** | Jesper Bojesen, M.Sc. Student<br>Anders Sewerin Johansen, M.Sc. Student<br>Morten Nicolaj Pedersen, M.Sc. Student<br>Maz H. Spork, M.Sc. Student<br>Martin Zachariasen, Post-Doc.<br>Hans-Henrik Stærfeldt, M.Sc. Student |
| **Academic partners:** | Martin Dietzfelbinger, Prof., Ilmenau<br>Torben Hagerup, Prof., Frankfurt<br>Jukka Teuhola, Senior Lecturer, Turku<br>Tomi Pasanen, Ph.D. Student, Turku<br>Peter Sanders, Ph.D., Saarbrücken |
| **Industrial partners:** | Jesper Larsson Träff, Ph.D., NEC, St. Augustin |


### Summary


In spite of faster computers, the tuning of programs may be necessary to make them useful in practice. Often programmers apply ad hoc hacks and tricks when tuning their programs. The goal in the core computer-science research discussed in this proposal is to understand better the principles behind these hacks and tricks. Our purpose is:

- to study the impact of the memory hierarchy of modern computers on the performance of programs,
- to explore how the parallelism in existence in modern computers can be used for improving the performance of programs,
- to make our programs for fundamental computational problems available as benchmark programs in the Worldwide Web, and
- to write high-quality lecture notes on performance engineering.


### Background

I, the undersigned, am a member of the algorithmics and optimization group at DIKU. The research activities of the group can be roughly divided into three categories: optimization, theoretical algorithmics, and experimental algorithmics. Most members of the group, like myself, are active in all these three areas. At the moment the focus of the group is in algorithm experimentation, especially for optimization problems. This work is sponsored by the SNF with a small grant that partially covers the conference travel of group members and the expenses of our short-term guests.

The project suggested requires substantial programmer resources. Ordinarily much of the work could be done as student projects, but because most of our students work full time or near full time in the industry, large programming tasks cannot easily be undertaken in this way. The main purpose of the application is to raise funds with which we can buy our students out from their current workplaces for a few months and hire them as programmers. As a desirable side effect, they can finish their studies and take actively part in the research carried out at DIKU.

Another badly needed resource is a secretary. The secretaries at DIKU are mainly involved in daily study administration whereas the group urgently needs a secretary for the following jobs: editorial correspondence, conference management, database management (bibliographies, homepages), technical typesetting (figures, tables), language checking, et cetera. The researchers are currently doing these tasks themselves. Having a secretary to do the jobs would free a large amount of research capabilities as well as make the group more visible.

We would also like to invite two visitors, Tomi Pasanen and Jesper Larsson Träff, to the group. Tomi is expert in performance engineering on sequential computers and Jesper that on parallel computers. Hence, they would be good trainees for our students. The support for these visitors will be applied, also from the SNF, with separate applications.

## Performance engineering

The main goal in performance engineering is to make programs faster. A typical situation is that one writes a program, or one would like to write a program, but back-of-the-envelope calculations show that a single run of the program would take unacceptably long time. In his book[1], Bentley gives a nice example about this: the original program (for the $n$-body problem) would have run about one year, but after performance engineering the final product was able to carry out the task in a day.

These kinds of improvements in performance are only achieved by working at many design levels. Often the most dramatic improvements are obtained at the algorithm and data-structure level. In Bentley's example a new algorithm reduced the execution time from one year to one month. Thereafter, the program may be improved by utilizing various algorithm-tuning, data-structure-tuning, and code-tuning techniques; and as a final resource one may run the program on a faster computer, if such is available.

The research conducted by myself, and my collaborators, is motivated by the computing aspects themselves, not by any particular application area. However, many of my colleagues working in areas like optimization, computer vision, robotics, et cetera now and then tune their programs. The emphasis in our research is to understand the algorithm-tuning, data-structure-tuning, and code-tuning techniques better. In particular, we take into account the hierarchy of memory levels and the parallelism prevalent in current high-performance computers. When the tuning principles are well understood, it might be possible to design automatic tools for the task. But it must be admitted that the tool building does not have a central role in our research at the moment.

## High-performance computers

The classical *random-access-machine* (RAM) model used in the analysis of algorithms consists of a processor and a one-dimensional memory that can be accessed in unit time. For two reasons the programs derived from algorithms proved to be optimal on a RAM may perform non-optimally on current computers: (1) due to the memory hierarchy, a memory access can take much longer than a computation accomplished by a processor; (2) by utilizing the parallelism that is in existence in various forms it might be possible to develop faster solutions. Below we discuss both of these issues in greater detail. For further details of modern computer organization, we refer to the book[2] by Patterson and Hennessy.

[1] Jon Bentley, *Programming Pearls*, Addison-Wesley Publishing Company, Reading (1986)

[2] David A. Patterson and John L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann Publishers, San Francisco (1994)

### Memory hierarchy

Most high-performance computers have several processors and a hierarchical memory, each memory level with its own size, speed, and cost characteristics. The processors have only access to the lowest memory level, the register file, so the data has to be there before it can be processed. The basic idea is to keep the regions of recently accessed memory as close to the execution unit as possible. If the memory reference cannot be satisfied by one memory level, i.e., in the case of a *miss* or a *fault*, the data must be fetched from the next level. The data is transferred in *blocks* between the different levels in the hope that the memory locations in vicinity of the current one will be accessed soon. The blocks have different names, *words*, *lines*, or *pages*, depending on the memory levels involved. Normally a miss at some memory level means that the processors must wait until the requested data has arrived in the registers.

The main characteristics of the different memory levels are listed in the following:

***Register file:*** The registers in the register file are multiported so that they can be accessed in parallel by several processing units in every clock cycle. The duration of a clock cycle is some nanoseconds and the size of the register file at most a few kilobytes.

***First-level cache:*** The first-level cache is still accessible every one or two clock cycles, but it has only few parallel ports. Currently, most first-level caches are quite small, 8–64 kilobytes, in order to be able to keep them on chip and close to the execution unit.

***Second-level cache:*** The second-level cache is considerably larger but has an order of magnitude higher latency. If it is off-chip, its size is mainly constrained by the cost of fast, random-access memory. Its typical size is one megabyte.

***Main memory:*** The main memory is built of high-density, low-cost, random-access memory. Including all overheads for cache misses, memory latency, and translation from logical memory addresses to physical memory addresses, a main-memory access can be two orders of magnitude slower than a first-level cache hit. The typical size of a main memory is 64 megabytes.

***Disk storage:*** A disk pack consists of one or more rotating *disks* and one *read/write head* per *disk surface*. On the disks the information is stored in concentric circles, called *tracks*, which are further divided into *sectors*. To read or write a sector, the read/write heads must be mechanically moved to the correct track, electronically switched to the correct surface, and then wait until the requested sector rotates below the heads. Due to the mechanical nature of disk-storage accesses, these are several orders of magnitude slower than main-memory accesses. Today the delay of accessing data from disk storage is several milliseconds.

Most computers have separate caches for data and program code, so the cost caused by instruction reads can be disregarded as long as programs are reasonably short. Therefore, the performance of a program is mainly determined by its memory access pattern when manipulating data.

### Parallelism

Parallelism is in existence in many different forms in current computers:

***Word-level parallelism:*** This denotes the ability of a computer to manipulate several bits stored in a word in one clock cycle. At the moment some of the computers use 64-bit words, but computers with a larger word size are under construction.

***Instruction-level parallelism:*** The instructions are executed in a pipelined fashion such that the execution of a new instruction is started before the execution of the previous one is finished. Conditional branches may be problematic since the instruction to be executed next is not known until at the end of the execution of such a branch. Also, a memory access might be problematic if its result is needed by one of the following instructions. In this environment programs that contain lot of independent instructions utilize pipelining best.

***Memory-level parallelism:*** One way to compensate the disparity between the slow memory and the fast processors is to furnish a computer with multiple disk packs that are used in parallel.

***Processor-level parallelism:*** Assuming that a computer has 32 processors, it might be possible to solve a problem 32 times faster than in a single processor system. However, in practice this optimal speed-up is only achieved for problems that are easy to parallelize, but in this case parallel computing really works, and it works well.

***Computer-level parallelism:*** A cluster of workstations could be seen as a virtual computer. In this environment the communication cost often dominates the overall costs. Hence, only problems whose solution requires very little communication should be attacked in this way.

## Planned research

As a concrete goal, we would like to provide a simple analytical model that can be used to estimate the performance of a program. There should be a close relationship between the estimate given by the model and the actual behaviour on a computer. However, it seems difficult to take into account all the technical details of current computers at the same time. Therefore, we have decided to make projections of the actual computer architecture and work on these approximations instead.

### Reducing instruction count

Traditionally, the performance is measured by counting the number of instructions or other primitive operations executed. The instruction count is significant for programs that are computation intensive, that do not use much memory, or that mainly access the memory sequentially. The performance of such a program can be improved by considering the inner loops; saving some instructions there will often speed up the program considerably. Inner-loop optimization has long traditions, many of the techniques still in use are mentioned in Knuth's paper[3] from 1971.

In consequence of the slowness of the memory accesses the algorithms developed for *sequential-access machines* (SAM), or Turing machines as they are also called, are useful even today. For instance, Schönhage et al.[4] observed that a SAM algorithm for multiplying two $n$-word integers with $O(n \log n \log\log n)$ work is actually faster than a RAM algorithm having $O(n)$ theoretical performance, when $n$ is large. Based on this observation they used the SAM model when implementing about 160 numerical algorithms.

To make a SAM program faster one should try to reduce either the number of sequential scans over the data or the number of the instructions carried out in a scan. To our surprise many classical programs are not well optimized in this respect. For example, we[5] were able to speed up a normal textbook implementation of mergesort by a factor of 4–6, depending on the quality of the textbook code. Our purpose is to prove analytically and verify experimentally similar results for other programs published in the computing literature. As an analytical tool in this work we use the pure C model introduced in our paper on mergesort.

### Utilizing word parallelism

Recently, many new algorithms have been developed for the so-called *word RAM* which is capable of manipulating words of $\lambda$ bits in constant time, where $\lambda$ is some positive integer. For

[3] Donald E. Knuth, An empirical study of FORTRAN programs, *Software—Practice and Experience* **1** (1971), 105–133

[4] Arnold Schönhage, Andreas F. W. Grotefeld, and Ekkehart Vetter, *Fast Algorithms: A Multitape Turing Machine Implementation*, B•I•Wissenschaftsverlag, Mannheim (1994)

[5] Jyrki Katajainen and Jesper Larsson Träff, A meticulous analysis of mergesort programs, *Proceedings of the 3rd Italian Conference on Algorithms and Complexity, Lecture Notes in Computer Science* **1203**, Springer-Verlag, Berlin/Heidelberg (1997), 217–228

a survey of the recent developments in the area, see the paper[6] by Hagerup. The crux of these new algorithms is to utilize various range-reduction techniques, pack several small integers into one word, and then manipulate these in parallel by exploiting the inherent parallelism of the word RAM.

For several reasons these algorithms are predominantly of theoretical interest. First, the subroutines operating with the packed integers require many integers to fit in one word in order to be effective. Second, the reductions themselves require lot of memory space. Third, the asymptotic time bounds derived seem to hide large constant factors.

Our aim is to investigate the usefulness of the word-RAM techniques. Can an algorithm utilizing some of the new ideas compete with the standard comparison-based methods, e.g., in sorting? In this comparison both the running time and the space usage should be taken into consideration. Furthermore, it would be interesting to know what happens when the word size $\lambda$ increases.

### Improving cache behaviour

On high-performance computers the cache-miss penalty is now over 100 clock cycles. This penalty is steadily increasing, according to a recent survey[7] 40 percent or more per year. Recent studies[8] show clearly that the performance of programs can be improved by taking into account cache effects. Therefore, our research focuses on the design and analysis of programs in the presence of caching.

More precisely, our goal is to increase the spatial and temporal locality in the existing programs, and to design new cache-conscious programs that perform better than existing ones. Programs that use pointers are especially interesting since often these perform poorly due to caching. At the moment we try to redesign some classical pointer-based algorithms that use heavily the random-access capabilities of the RAM model. In this work a clear motto is that one should use arrays instead of pointers.

### Improving paging performance

The memory overhead is a dominant factor in the execution time of many programs. This is especially true for programs that process massive amounts of data, i.e., more than what can be placed in main memory. In this case the performance of a program is determined by its paging behaviour, i.e., the number of pages transferred between main memory and disk storage.

Normally a page can reside in any place in main memory. In the case of a page fault, one of the pages in main memory has to be moved to disk storage. Usually this page replacement is controlled by operating systems, but some systems provide an application-controlled management of the pages in main memory. Our target is just these systems. Recently, we[9] proved improved theoretical complexity bounds for the maintenance of a priority queue on disk storage. Our

[6] Torben Hagerup, Sorting and searching on the word RAM, *Proceedings of the 15th Annual Symposium on Theoretical Aspects of Computer Science*, *Lecture Notes in Computer Science* **1373**, Springer-Verlag, Berlin/Heidelberg (1998), 366–398

[7] Alvin R. Lebeck and David A. Wood, Cache profiling and the SPEC benchmarks: A case study, *Computer* **27**,10 (1994), 11–26

[8] Anthony LaMarca and Richard E. Ladner, The influence of caches on the performance of heaps, *The ACM Journal of Experimental Algorithmics* **1** (1996), Article 4. Available from `http://www.jea.acm.org/`

Anthony LaMarca and Richard E. Ladner, The influence of caches on the performance of sorting, *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, ACM, New York and SIAM, Philadelphia (1997), 370–379

[9] Ramzi Fadel, Kim V. Jakobsen, Jyrki Katajainen, and Jukka Teuhola, Heaps and heapsort on secondary storage, *Theoretical Computer Science*, to appear

Gerth Stølting Brodal and Jyrki Katajainen, Worst-case efficient external-memory priority queues, *Proceedings of the 6th Scandinavian Workshop on Algorithm Theory*, *Lecture Notes in Computer Science* **1432**, Springer-Verlag, Berlin/Heidelberg (1998), 107-118 purpose is to attack other fundamental data-processing problems for which the best possible resource bounds are not yet known.

### Utilizing computer parallelism

Our primary goal in this area is to develop a distributed shared-memory system that would perform better than existing ones. In our opinion the analysis and the description of existing implementations are unsatisfactory. Moreover, it is unclear for us why the performance of these systems is so poor. Our secondary goal is to test how the developed system could be used in the development of software for number crunching on the cluster of (Alpha) workstations available in our department.

København, 25 September 1998

Jyrki Katajainen
Assoc. Prof., Docent, Ph. D.