Research proposal: software tools for program library development

Institution:	Department of Computing, University of Copenhagen (DIKU)
Proposed duration:	1.1.2002 – 31.12.2004
Principal investigator:	Jyrki Katajainen, Assoc. Prof.
Other investigators:	Dennis Haney, B.Sc. Student
	Bjarke Buur Mortensen, M. Sc. Student
	Lars Yde, M. Sc. Student
Academic partners:	Gerth Stølting Brodal, Assoc. Prof., Århus
	Tomi Pasanen, Lecturer, Turku
	Peter Sanders, Dr. rer. nat, Saarbrücken
Industrial partners:	Jesper Bojesen, M. Sc., Medical Insight, Copenhagen
	Sofus Mortensen, B. Sc., lambdasoft, Copenhagen
	Jesper Larsson Träff, Ph. D., NEC, St. Augustin

Summary. A *program library* is a collection of off-the-shelf components used as building blocks in other programs. With such libraries larger applications can be developed faster and more cost-efficiently. The Standard Template Library (STL) is part of the ISO standard[1] for C++ which was ratified in 1998. C++ is one of the most widely used programming languages, and the STL components are expected to become some of the most widely used software in existence. The effort to develop a program library is often underestimated. Kurt Mehlhorn, who is one of the main architects of the LEDA library[2], estimated that the LEDA project would be a one-year project, but its development took ten years. One reason for this unfortunate situation is the lack of software tools supporting program library development. In the Copenhagen STL (CPH STL) project[3] our goal is to develop such tools. In brief, the goals of the project are:

- develop software tools that can be used in the development of component libraries,
- provide an enhanced edition of the STL and make it freely available on the Internet,
- provide cross-platform benchmark results to give library users better grounds for assessing the quality of different STL components,
- carry out experimental algorithmic research, and
- expose our senior graduate students to project work similar to what they may encounter in their future work as professionals.

[1] International Organization for Standardization (ISO), *ISO/IEC 14882: Standard for the C++ Programming Language* (1998).
[2] K. Mehlhorn and S. Näher, *LEDA: A Platform for Combinatorial and Geometric Computing*, Cambridge University Press (2000).
[3] `www.cphstl.dk`

CPH STL Report 2001-15, September 2001.

Background

During the next three years the CPH STL project will have the highest priority in the activities of the Performance Engineering Laboratory (PE-lab) led by the principal investigator at the Department of Computing, University of Copenhagen. Our primary motivations for developing the library were:

1. Our earlier research[4] pointed out that the efficiency of some of the components in existing versions of the STL could be improved considerably. Hence, it was natural to start a more extensive investigation about the efficiency of various STL components.
2. Existing versions of the STL provide only one implementation for each component whereas we want to provide several alternative implementations. By providing benchmark results on the efficiency of these alternatives a library user can better assess which alternative is suitable for his/her purposes.
3. The STL may be extended in many different ways. Its extension to new areas provides challenging research problems both at the design and algorithm level.
4. The STL is a marvelous teaching tool. The members of the PE-lab have already used the library in four graduate courses:
   - The CPH STL: weekly team meetings, Autumn 2000
   - Performance engineering, Spring 2001
   - The CPH STL: weekly team meetings, Spring 2001
   - My favourite software development tools, Autumn 2001

   We expect that after a maturation period the library, and its infrastructure, will be integrated to graduate courses in other universities, too.

We have divided the development of the CPH STL into three phases (see the time line in Figure 1). In Phase 1, which we have just finished, the problem area was analysed and the development tools were selected[5]. Up to now ten student projects have been finished under the umbrella of the CPH STL, and prototypes for the most important components of the library exists. Phase 1 was partially supported by Danish Natural Science Research Council under contract 9801749 (project Performance Engineering).

With this application we seek support for Phases 2 and 3 of the project. After one year of development, it is clear that innovative software tools are critical success factors when transferring the existing prototypes to a product. Therefore, in Phase 2 we put emphasis on the creation of software

[4] Jesper Bojesen, Jyrki Katajainen, and Maz Spork, Performance engineering case study: heap construction, *The ACM Journal of Experimental Algorithmics* (to appear).
Jesper Bojesen and Jyrki Katajainen, Interchanging two segments of an array in a hierarchical memory system, in *Proceedings of the 4th International Workshop on Algorithm Engineering, Lecture Notes in Computer Science* **1982**, Springer-Verlag (2000), 159-170.

[5] See, the minutes of the team meetings available at `www.diku.dk/undervisning/2000e/e00.505/`.

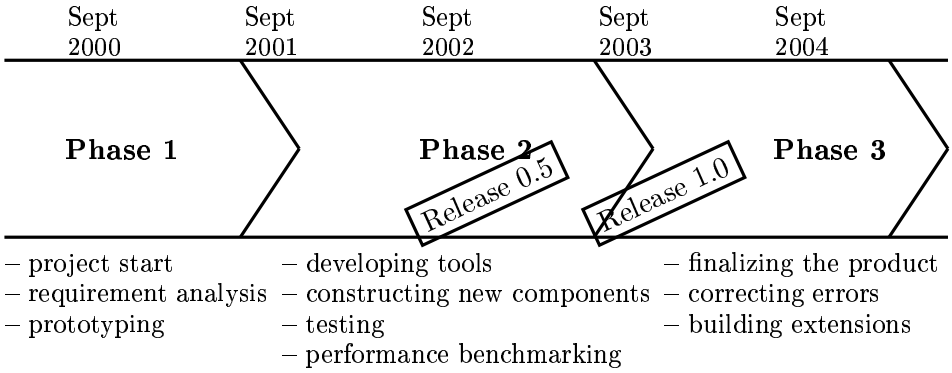

Figure 1. Time line for the CPH STL project.

tools supporting program library development. Our hope is that the tools could be used in the development of other libraries written in C++ as well. We provide a beta release of the CPH STL in September 2002, and a stable release in September 2003. In Phase 3 we finalize the library and consider possible extensions.

To strengthen the infrastructure of the project, one research assistant is needed. This position is not intended for full-time employment of the same individual for a three-year period. Rather, it will be filled by senior graduate students contractually employed for a period of six months which will hopefully stimulate the research environment. Other budget items are: costs associated with participation in conferences, expenses borne by our academic and industrial partners, and the purchase of two workstations and relevant literature. The main body of the programming work is expected to be done by the students associated with the PE-lab.

Lessons learnt in Phase 1

Originally we were mainly interested in the algorithmic aspects of the STL, but in Phase 1 it became clear that language aspects and software-development aspects are important, too. Let us briefly consider two of the many relevant language aspects.

First, C++ was an extension of the C language, but a new C standard was ratified in 1999, so C is no more a subset of C++. In our opinion the evolution of the C language is going in a wrong direction, so we would like to see a C++ language without C. Hence, we will develop class templates `integer` and `real` that could be used to replace the normal C types `short`, `int`, `long`, `float`, and `double`. Currently, many compilers have efficiency problems with small objects[6] so it would interesting to know what can be

[6] This problem is discussed in [A. D. Robison, The abstraction penalty for small objects in C++, in *Proceedings of the 2nd Parallel Object-Oriented Methods and Applications Confer-*

done for this problem.

Second, the syntax of C++ can be awkward, especially when templates are used in the form popularized by Andrei Alexandrescu[7]. In a sense C++ is a two-level language: a program describes the *usual computation*, where values are combined to produce other values, and the *type computation*, where types are combined to produce other types. Our hope is that we can keep many of the cryptic template constructs in the library, and hide them from novice users. In long term the syntax of the type-computation level of the language should be made clearer.

Even though the algorithmic and language aspects are important, the software-development aspects of the project are far the most important in order to finish the project in reasonable time. Efficient programs can sometimes be ten times longer than straightforward textbook solutions. Often they involve several special cases which make the control structure complicated and will hence jeopardize the correctness. Also, in different subprojects we have repeatedly been carrying out the same tasks, so more tools common to the whole team must be developed.

Tools to be developed in Phase 2

The CPH STL project is an open-source project and the library is available — even in its incomplete form — on the Internet. Due to this open-source character we are mainly relying on open-source software-development tools. However, in our project we need some tools specific to library development that are not currently available, or at least a significant amount of development is needed for adapting existing tools for our purposes.

Cross-platform benchmarking. In addition to the program library we want to provide cross-platform benchmark results on the efficiency of various library components in different execution environments taking into account the variation in compilers and computers. This means that several hundred benchmarks must be carried out. Preparing the programs and scripts needed for even one such benchmark can take days of development time. However, much of this work can be automated and the vision is to create an automatic benchmarking tool that would be useful in any experimental algorithmic work. A developer should only fill a form telling which programs to benchmark, on which data, on which computer, and with which compiler, after which the system must be able to provide the benchmark results in a graphical form.

Another problem in benchmarking is the imprecision of the timing facilities provided by many computers. Koenig and Moo[8] described an interesting

ence (1996)] which is available at `www.acl.lanl.gov/Pooma96/abstracts/robison.html`.

[7] Andrei Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley (2001).

[8] Andrew Koenig and Barbara E. Moo, Performance: myths, measurements, and morals. Part 6: useful measurements—finally!, *Journal of Object-Oriented Programming*, May 2000, 30–33.

solution to how the clock precision can be improved from milliseconds to nanoseconds. We will incorporate their idea into our tool since it makes mass benchmarking computationally feasible.

Automatic program transformations. In Phase 1, we observed that most STL components can be sped up by unrolling the loops. Due to the complexity of C++, most compilers cannot do this optimization, so it must be done by hand. This makes the programs less readable and requires careful programming. Since the underlying program transformation is regular, the process can be automated. Our purpose is to write scripts that carry out this optimization, so that we can keep the library routines readable. By making the scripts publicly available, we hope that next generation C++ compilers are able to perform this optimization for us.

Pure-C translator and profiler. Library routines must be well tuned, and one of the parameters one has to consider is the number of instructions executed. Our meticulous analysis of programs is based on the pure-C language[9]. The model is machine-independent, and the pure-C instruction count reflects reasonably well the actual runtime behaviour when CPU-cost is predominant. The meticulous analysis of programs is tedious and therefore we need a compiler that translates C++ programs into pure C. It would not be feasible to build a C++ compiler from scratch, but it should not be difficult to modify an existing compiler (e.g., GNU g++ compiler) so that the machine instructions produced by it are replaced with pure-C instructions.

Above such a compiler, a profiler can be built which measures the number of instructions executed by a program for a typical input. A similar typical-case analysis was used in the numerical library development led by Schönhage[10]. In their project the underlying machine model was a sequential-access machine (Turing machine).

Cache profilers. Cache-efficiency is one of the algorithmic aspects ignored in existing STL implementations. At the moment algorithmic community is interested in cache-oblivious data structures and algorithms[11], i.e., methods that perform efficiently at all memory levels even though the size of caches and cache blocks is not known. The research on the practical utility of cache-oblivious methods has just started[12], and our goal is to continue

[9] Pure C was originally defined in [Jyrki Katajainen and Jesper Larsson Träff, A meticulous analysis of mergesort programs, in *Proceedings of the 3rd Italian Conference on Algorithms and Complexity*, *Lecture Notes in Computer Science* **1203**, Springer-Verlag (1997), 217–228], and it was further refined in [Jesper Bojesen, Jyrki Katajainen, and Maz Spork, Performance engineering case study: heap construction, *The ACM Journal of Experimental Algorithmics* (to appear)] and in [Sofus Mortensen, *Refining the pure-C cost model*, M. Sc. Thesis, Department of Computing, University of Copenhagen, (2001)].

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

[11] See, for example, [Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob, Cache-oblivious search trees via trees of small height, in *Proceedings of the 13th Annual Symposium on Discrete Algorithms*, ACM-SIAM (to appear). A draft is available at `www.daimi.au.dk/~gerth/emF01/papers/BFJ01.ps.gz`.

[12] Naila Rahman, Richard Cole, and Rajeev Raman, Optimized predecessor data structures for internal memory, in *Proceedings of the 5th Workshop on Algorithm Engineering*,

this line of research for the STL containers.

Several computer vendors provide cache profilers specific to their own products. Usually these utilize hardware registers and therefore profiling will only cause a small extra overhead. In our group we have already developed two general-purpose cache profilers[13] that make cross-platform cache profiling possible. However, both of these profilers have turned out to be too slow for production purposes. A fast general-purpose cache profiler, making cross-platform cache profiling possible, is still in our wish list.

Concept checking. Two main innovations of the STL are generic programming and programming with concepts. A *concept* is a set of requirements that a type — normally a template parameter — must fulfil. All the concepts behind the STL components are nicely listed in the book by Austern[14]. However, the C++ language itself does not provide any mechanism for handling concepts, but concept checking has to be done by the programmer. Furthermore, all the concept checks must be carried out in compile time.

Both the SGI STL[15] and the Boost libraries[16] provide tools for concept checking. The foundation of these tools lies in the type-computation level of C++ so the programs are complicated. Due to the complex language structures involved and imperfect compiler support, the design, implementation, and verification of concept checks is a time-consuming and error-prone process. Library developers would benefit greatly if some or all of this process could be automated.

Integrated testing environment. One of the innovations done in the development of the LEDA library was that programs carrying out complicated computations should test their results if possible. The concept is called *self-testing*. This approach is against the traditional efficiency principle which is one of the foundations of the STL. In spite of this, we think of that one should apply self-testing when possible.

Modern imperative programming has many other pitfalls; here we think of exception handling and memory management. The development of exception-safe and memory-safe code is demanding. We need tools for confirming the exception safety of our code and tools for checking that the programs do not have memory leaks. That is, developer support is required so that it would be easier to produce reliable software. A few sporadic tools exist but a significant amount of scripting is needed to create an integrated testing environment.

Lecture Notes in Computer Science **2141**, Springer-Verlag (2001), 67–78.

[13] See `www.diku.dk/research-groups/performance-engineering/resources.html`.

[14] Matthew H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, Addison-Wesley (1999).

[15] `www.sgi.com/tech/stl/`

[16] `www.boost.org/`

Budget

Budget item	First year	Three years
research assistant	275 000 DKK	825 000 DKK
visitors	20 000 DKK	60 000 DKK
travelling expenses	20 000 DKK	60 000 DKK
two workstations	50 000 DKK	50 000 DKK
literature	5 000 DKK	15 000 DKK
Sum	370 000 DKK	1 010 000 DKK
overhead 20%	74 000 DKK	202 000 DKK
In total	444 000 DKK	1 212 000 DKK

On behalf of the project group

Copenhagen, 30 September 2001

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