Performance Aspects of a DSEL for Scientific Computing with C++

R. Heinzl△, P. Schwaha△, M. Spevak°, and T. Grasser△

△Christian Doppler Laboratory for TCAD in Microelectronics
at the Institute for Microelectronics

°Institute for Microelectronics, Technical University Vienna,
Gußhausstraße 27-29/E360, A-1040 Vienna, Austria
E-mail: heinzl@iue.tuwien.ac.at

Abstract
In the field of scientific computing there is a manifold of software applications and tools available which provide methods and libraries for the solution of very specific problem classes [1, 13, 15, 22, 26]. They are mostly specialized on a certain type of underlying mathematical model resulting in a solution process which is highly predictable. We have designed a topological library [11] which enables the development of an equation specification mechanism for all different kinds of discretized partial differential equation by utilizing functional programming. This work examines the performance aspects of our domain specific embedded language (DSEL) approach for mathematical equation specification.

1. Introduction

Our institute has gained profound experience in developing and releasing scientific code [4, 10, 19]. The software products in the list below represent the chronological output of the institute's research and development activities over more than a decade (OO: object oriented):

`S*AP`	Two- and three-dimensional interconnect simulator (imperative)
`MINIMOS-NT`	Two- and three-dimensional device simulator (imperative, OO)
`SIESTA`	Framework for various simulators and optimization (OO)
`FEDOS`	A three-dimensional PDE solver (OO)
`WSS`	Object-oriented geometrical and topological library (OO)

With this steady evolution of software tools, the shift and change of programming paradigms can be clearly observed. In the first years at our institute, only one and two-dimensional data structures with so called structured topology cell types were used due to the limited resources of the computer. The structured programming paradigm was sufficient for this type of task [10]. With the improvement of computer hardware and the developing object-oriented programming paradigm, the shift to more complex data structures such as unstructured three-dimensional simplex and cuboid cell complex types was possible. The development of complex applications for process [4] and device [14] simulation are examples for this evolutionary path. All of the applications developed at our institute use very specialized data structures for describing triangles, quadrilaterals, tetrahedra, cuboids with different mechanisms for access, traversal operations, and data storage. The most basic reason for this is the fact, that due to the assembly of large linear equation systems, any overhead from generic data structures would increase the long simulation run-times drastically.

Nowadays, the simulation requires different dimensions, different topological cell complex types, different numerical treatment, orthogonal module usage, and numerical methods employed in discretization, interpolation, or optimization. These methods make use of highly nonlinear functions $f_x(\mathbf{x})$ which can consist of several coupled differential equations. Therefore the field of scientific computing requires an efficient and sufficient notation of equation systems, has to construct equations and has to abstract the topological traversal of different underlying objects. By enforcing a unique and generic interface to all different kinds of topological cell types and a new type of equation specification mechanism for all different kinds of discretized partial differential equation utilizing functional programming is made possible. Therefore we have developed a DSEL approach for a mathematical equation specification based on a generic topology library [11, 25].

This work examines the performance aspects of our DSEL with respect to runtime on various computer systems and the influence of different compiler versions. The language of choice is C++ due to the possible high execution speed of the compiled code [26] and the support of different programming paradigms, such as object oriented, generic (GP), functional (FP [5, 6, 17]), and meta-programming (MP [2]). It has also been shown that C++ supports the development of efficient and high performance libraries capable of handling complex topics such as graph treatment [8, 9, 22] unlike pure functional programming languages like Haskel or ML.

2. Motivation

The performance of computer systems with respect to a given set of applications depends on numerous factors and can not be attributed only to the speed of the central processing unit (CPU). Among the most important factors is the connection of the CPU to the computers main memory [7]. Naturally the focus of the evolution of CPUs was to increase their processing speed. This goal was greatly aided and in fact only made possible by continuous downscaling of the dimensions of the devices are built on. This downscaling of the densely packed logic found in CPUs made it possible to attain ever higher clock speeds thereby increasing their maximum performance. The main effect for random access memory (RAM) modules, on the other hand, was to increase their sizes, again by an ever growing level of integration.

This trend of increasing clock speeds, especially of CPUs, has already led to the problem that CPUs require data at a faster rate than memories are able to supply it. This has resulted in the development of memory hierarchies introducing several levels of caches and instruction pipelines, thereby increasing the overall performance of the systems.

From a software point of view, new paradigms such as OO, GP, FP, and MP are now available and can even be used together. The non-trivial and highly complex scenario of scientific computing yields itself exceptionally well to the combination of these different programming paradigms with respect to their advantage. The features of meta-programming offer the embedding of domain specific terms and mechanisms directly into the language as well as compile-time algorithms to obtain an optimal run-time. The generic programming paradigm establishes homogeneous interfaces between algorithms and data structures without subtyping polymorphism. Functional programming eases the specification of equations and offers extendable expressions while retaining the functional dependence of formulae due to higher order functions.

The unique way parametric polymorphism is realized in C++ [18] makes it possible to write compile time libraries, that enable an optimization across the boundaries of these libraries, thereby reaching new performance optima, at the expense of increased compile time. This has already been demonstrated in the field of numerical analysis yielding figures comparable to Fortran [15, 26], the previously undisputed candidate for this kind of calculations.

Current compilers have to utilize all these features to generate the most efficient code.

Generic Library Approach

First, we will focus on the advantages of a generic topology library that includes all classes and algorithms for the most important topological cell complex types such as simplex and cuboid types.

This library is based on all the requirements extracted from the applications developed at our institute. It greatly reduces the applications development and the information necessary to construct them. It accomplishes this by storing topological data and entering a formal interface specification [12, 25]. It provides incidence traversal and orientation operations with a generic interface similar to the C++ STL. Therewith algorithms and complete discretization schemes can be specified orthogonally without specifying the actual topological structure. Problems which are hard to adapt to existing libraries due to the resctrictions of the underlying data structures, can be handled easily using this library.

The second important part is based on a functional specification library [12, 20] which implements the basic features for a mathematical domain specific language. The specification of equations with automatic derivations by means of truncated Taylor series as well as discretized differential operators are the major parts for this library. The library offers support for different discretization schemes embedded into a functional specification language which offers higher-order functions. Therewith algorithms and equations are specified independently of the actual dimension, topology, and numerical data type of the cell complex type. We have already shown with simple equations such as the Laplace equation as well as some more sophisticated examples (e.g. the drift-diffusion semiconductor equations) [11] that our approach overcomes most of the dimension-dependent and data structural problems.

To briefly introduce our DSEL approach for scientific computing, we present an example from the users point of view. A typical representative of a 0-cell complex is the topological structure of a simple array. The C++ STL containers such as vector or list are representatives and are schematically depicted in Figure 1 . The points represent the cells on which data values are stored whereas Figure 2 presents a 1-cell complex type with different dimensional traversal mechanisms. The red arrows highlight the actual topological traversal mechanisms.

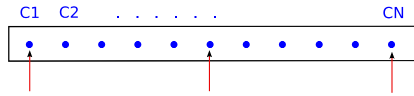

Figure 1: Topological traversal of vertices.

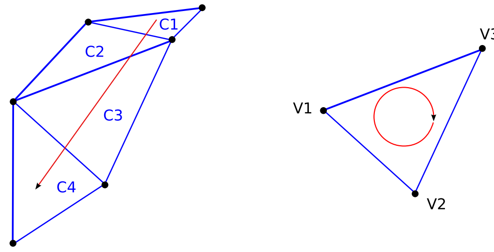

Figure 2: Topological traversal of cells and incident vertices of a cell.

Based on this traversal mechanisms, a transformation of a simple discretized equation to C++ code (DSEL) is straightforward and is explained in more detail in the next section. Here we present an example of the Poisson equation with a finite volume discretization scheme [16, 21]. This equation can be formulated as:

Poisson equation with our DSEL

```
equation =
(
  sum<vertex_edge>
  [
    diff<edge_vertex>[pot] * A/d *
    sum<edge_vertex>[epsilon] / 2
   ] - q
)(vertex);
```

The term `pot` represents the distributed potential, `A` the Voronoi area, `d` the distance of two points from each other, `q` the charge, and `epsilon` the permittivity. It is important to stress that all quantities have to be evaluated in their proper context (data locality), that is `pot`, `epsilon`, and `q` on vertices and `A`, `d` on the corresponding edges.

3. DSEL Approach

This section presents the topological traversal approach and the DSEL mechanism in more detail. The main aspect of our DSEL in C++ is the achievement of an efficient way to describe complex equations supported by a comprehensive topological traversal mechanism. By formalizing the topological access to different cell complex types, the DSEL mechanism offers itself to specify expressions and even complete equations in C++ directly.

Topological Traversal

To map a continuous problem to the regime of finite computations space must be divided into a finite cell complex $\mathcal{C}$. Our topology approach extends the C++ STL standard containers which can be represented as a 0-cell and the topological structure of a graph which can be represented as a 1-cell complex type to a multi-dimensional cell complex type with the following geometrical representations for the corresponding cells:

- 0-cell: vertex
- 1-cell: edge
- 2-cell: triangles, quadrilaterals
- 3-cell: tetrahedra, cubes
- 4-cell: hyper-tetrahedra, tesseract

The iterator concept, which is one of the key elements of the C++ STL, separates the actual access to data structures from algorithms. On the other hand, it combines iteration (topological traversal) and data access. This fact complicates the classification of iterator types and value access requirements [3].

Whereas graph algorithms use vertices and edges to store quantities, the numerical methods employed in the field of scientific computing require a more general approach. Scientific computing strongly depends on the arbitrary *location of data* which means that quantities can be attached to arbitrary objects such as vertices, edges, or triangles. Our topological traversal [3] is completely separated from actual access to data and is used as the underlying mechanism to obtain consistent access to data. This separation takes care of several well known problems of C++ [3] and suits itself well for the use of lightweight objects [23] as it offers great opportunities to store various quantities on all topological objects.

The following code snippet shows the traversal of a data structure with the topological structure of a 1-cell complex (graph). The separation of topological traversal and data access can clearly be observed. The traversal of the attached vertices to a cell can also be observed at the bottom part of the code snippet. We use the well known name *iterator* for the traversal mechanism instead of the proposed name *cursor*.

Generic topological traversal

```
typedef topology<cv_container,simplex,1> t_t;
topology_traits<t_t>::iterator          it;
typedef quantity<scalar, double>        quan_t;
quantity_traits<quant_t>::value         value;

t_t  container;
it = container.vertex_begin();
quan_t quan(container,"key");

++it;                  // traversal
value = quan(*it);     // access
cell_on_vertex_iterator covit(*it);
while(covit.valid())
{
  // operate on cell
  ++covit;
}
```

Equation Specification

Our approach takes care of accessing quantities of different locality e.g. quantities on vertices, edge, facets, or cells. The access mechanism, based on a property map concept [3, 24] is initialized with a domain, which is an instance of a cell complex. During initialization, the property map `pot` is bound to a specific domain with its quantity key. In the following example, `pot` is simply assigned a value:

Quantity assignment

```
string key_pot = "MyBuiltInPotential";
quan_t pot = scalar_quan(domain, key_pot);

for_each ( domain.vertex_begin(), domain.vertex_end(),  pot = 1.0 );
```

Based on the functional programming approach supported by the Boost phoenix library [6] a simple specification of equations is possible. With our topological traversal mechanisms and generic functions the Laplace equation can be specified in the following way:

Laplace finite volume discretization

```
for (vit = container.vertex_begin(); vit != container.vertex_end(); ++vit)
{
   equation = (sum<vertex_edge>
   [
     diff<edge_vertex> [pot]
   ])(*vit);
}
```

The `sum` and `diff` represents generic functions specialized for the field of scientific computing. The complete equation is examined at compile-time to construct the complete structure of the equation. The complex resulting from this mapping is completed by specifying the currently iterated vertex objects `*vit` at run-time which clearly demonstrates the compile-time and run-time border.

4. Performance Analysis

This section analyzes the performance behavior of our DSEL for topological traversal and equation specification. First, the topological traversal mechanisms are compared with a 0-cell complex implementation without any generic approach and a 1-cell complex type with the Boost graph library.

For the functional equation specification, the Blitz++ benchmark is used to estimate the overall performance behaviour of our computer systems. Afterwards, simple vector addition are compared on different computer systems and with different compiler versions. The influence of the actual computer system as well as the compiler used with the corresponding optimization switches are presented.

To show the maximum of achievable performance different computer systems are summarized in the following table showing the CPU types, amount of RAM, and the compilers used. A balance factor (BF) is evaluated by dividing the maximum MFLOPS measured by ATLAS [27] by the maximum memory band width (MB) in megabytes per second measured with STREAM [7]. This factor states the relative cost of arithmetic calculations compared to memory access.

CPU type	RAM	GCC	MFLOPS	MB/s	Balance factor
P4, 2.8 GHz	2GB	4.0.3	2310.9	3045.8	0.7
AMD64, 2.2 GHz	2GB	3.4.4	3543.0	2667.7	1.2
IBMP655,8x1.5 GHz	64GB	4.0.1	16361.7	4830.4	3.4
G5, 4x2.5 GHz	8GB	4.0.0	24434.0	3213.3	7.6

Topological Traversal

To investigate the abstraction penalty of our generic code we analyze a simple C implementation without any generic overhead compared to our topological environment. The next code snippet presents the C source code for a three dimensional 0-cell complex (cube):

C approach for 0-cell traversal

```
for (i3 = 0; i3 < sized3; i3++)
  for (i2 = 0; i2 < sized2; i2++)
    for (i1 = 0; i1 < sized1; i1++)
    {
       //operations
       // use i1, i2, i3
    }
```

Generic approach for 0-cell traversal

```
vit1 = container.vertex_begin();
vit2 = container.vertex_end();

for (;vit1 != vit2; ++vit1)
{
  //operations
  // use *vit1
}
```

Figure 3 presents results of four computer systems. As can be seen, the performance is comparable on all different systems without incurring abstraction penalty of the highly generic code compared to a simple C implementation.

To provide a more complex example, we compare the traversal mechanisms of the BGL data structures to our own approach. The following code snippets present the corresponding implementations.

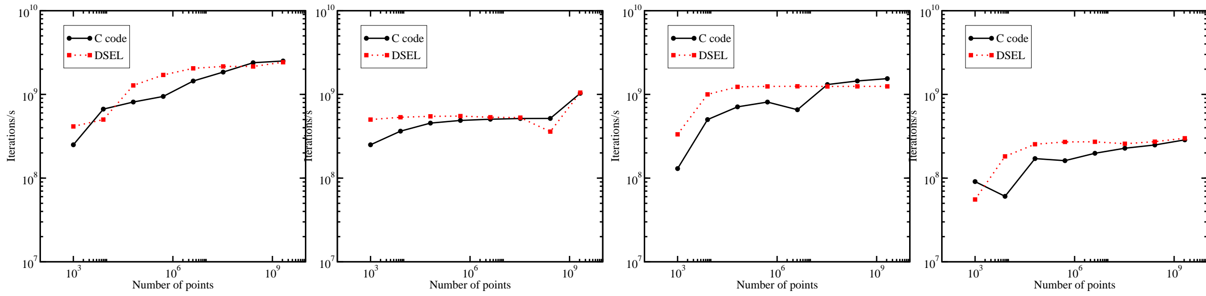

Figure 3: 0-cell traversal on the following computer systems (from left to right): P4, AMD64, G5, and IBM.

BGL incidence traversal

```
typedef adjacency_list<vecS, vecS> Graph;
Graph gr(number_of_points);
// edge initialization
graph_traits<Graph>::edge_iterator ei, ei_end;

for (tie(ei, ei_end) = edges(gr);
     ei != ei_end; ++ei)
{
  test_source1 += source(*ei, gr);
  test_source2 += target(*ei, gr);
}
```

Generic approach for incidence traversal

```
typedef topology<cv_container,simplex,1> t_t;
t_t container(number_of_points);
// cell initialization

typedef topology_traits<t_t>::cell_on_vertex_iterator  covit_t;
covit_t covit = covit_it(*container.vertex_begin());

while(covit.valid())
{
  test_source1+= source(*covit, container);
  test_source2+= target(*covit, container);

   ++covit;
}
```

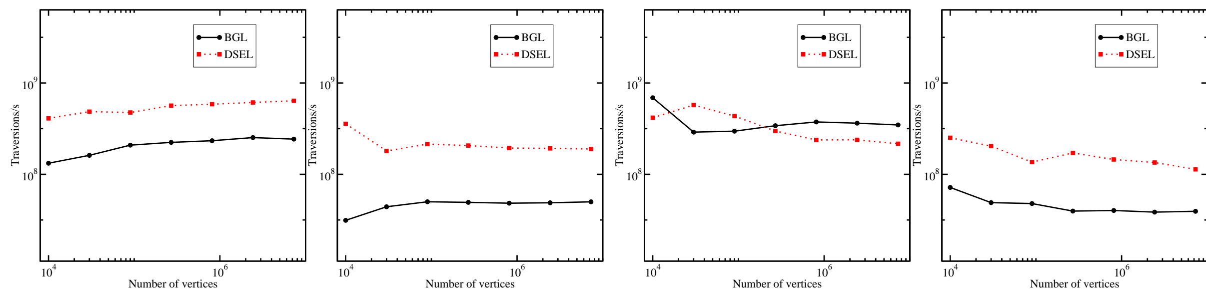

Figure 4: Incidence traversal for BGL and our approach on the following computer systems (from left to right): P4, AMD64, G5, and IBM.

Figure 4 presents the run-time results for edge on vertex traversal. The overall run-time behavior of our approach is comparable to that of the BGL.

Equation Specification

To compare different equation specification approaches we analyze several techniques which are available in C++. All approaches are compared to a hand-optimized Fortran 77 implementation:

- Naive C++ implementation with `std::vector<T>`
- Blitz++ Version 0.9
- Simple version of expression templates [2]
- C++ `std::valarray`
- Boost phoenix library (our DSEL)

The following compiler switches were used:

```
AMD64: -O3 -march=k8 -funroll-loops  -mtune=k8  -fforce-addr
P4: -O3 -march=pentium4 -funroll-loops -mtune=pentium4  -mfpmath=sse -ffast-math
G5: -O3 -mcpu=G5 -funroll-loops  -mtune=G5
IBM: -O3 -funroll-loops -mcpu=powerpc64 -maix64 -pipe
```

The test is performed using a vector addition $A_f = A_b + A_c + A_d$, evaluated with different vector sizes. Figure 5 compares the different approaches.

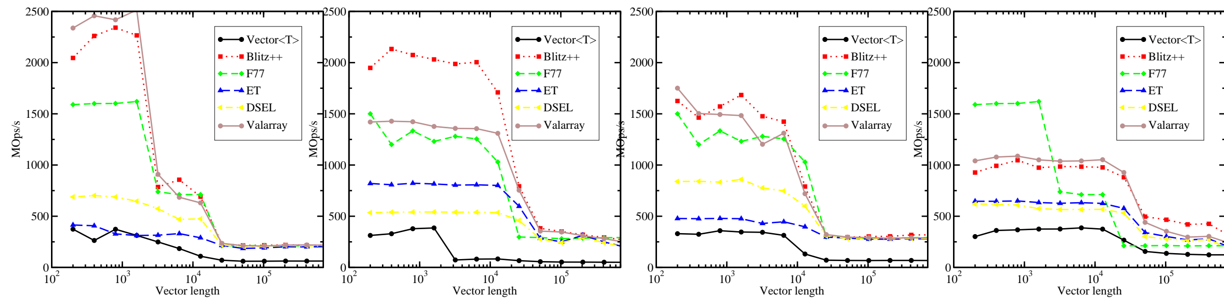

Figure 5: Performance of the evaluated expression on the following computer systems (from left to right): P4, AMD64, G5, and IBM.

For vector lengths smaller than 10^4, cache hits reveal the full computation power of the CPU, longer vectors show the limits imposed by memory band width.

To analyse the influence and ongoing advancements in compiler technology, we have analysed different compiler versions. Therewith the influence on different machines can be clearly separated into the system related differences and the compiler based differences. As a base foundation we use the Blitz++ benchmark system which uses a vector operation (DAXPY) $f = \alpha \cdot x + y$, evaluated with different vector sizes.

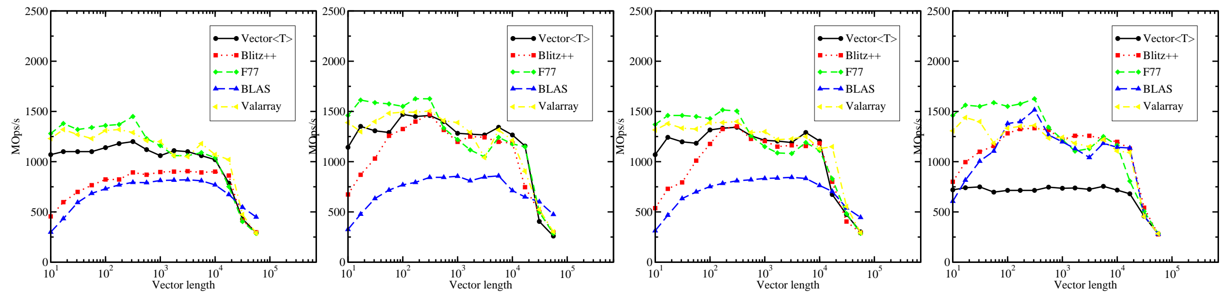

Figure 6: Blitz++ benchmark with different compiler versions: GCC-4.0.3, GCC-4.1.0, GCC-4.1.1, GCC-4.2.0

Finally, we compare our vector addition benchmark on the P4 with different compilers.

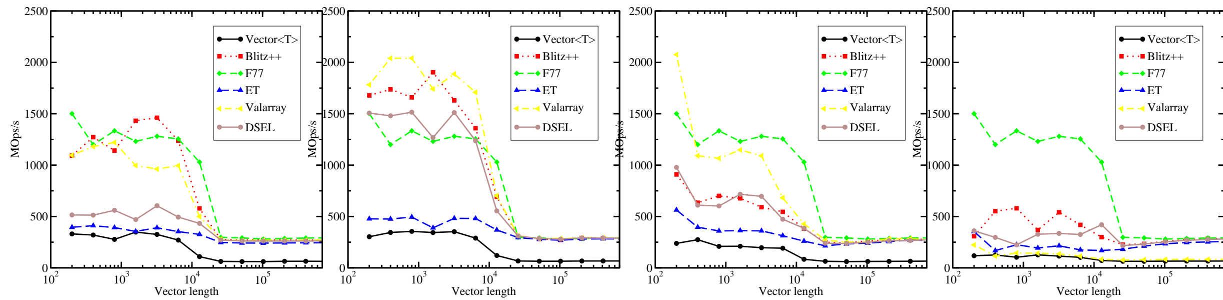

Figure 7: Different compiler versions on the P4: GCC-4.0.3, GCC-4.1.0, GCC-4.2.0, Intel 9.0

As can be seen, the functional approach based on the Boost phoenix library performs well with recent compilers such as the GCC-4.1.0 compiler generation. GCC-4.2.0 is still being developed at the moment and only presented to show the possibilities of the emerging compiler generations. The Intel 9.0 compiler does not perform well with all different specification mechanisms.

5. Conclusions

While computer performance is steadily increasing the additional complexity of simulation models easily outgrows this gain in computational power. It is therefore of utmost importance to employ the latest techniques of software development to obtain high performance and thereby ensure adequate simulation times even for complex problems. We created an infrastructure, that supports the formulation of different problems in a highly expressive way.

The combination of high expressiveness at such an excellent performance characteristic is only possible by using all the facilities provided by C++. Currently no other language offers sufficient support for all the necessary programming paradigms to enable this high level abstraction at this run-time speed.

6. Acknowledgements

We express our gratitude for the valuable discussions and the continuing support given by Prof. Siegfried Selberherr.

References

[1] A. Fabri. CGAL- The Computational Geometry Algorithm Library, 2001. `citeseer.ist.psu.edu/fabri01cgal.html`.

[2] D. Abrahams and A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[3] D. Abrahams, J. Siek, and T. Witt. New Iterator Concepts. Technical Report N1477 03-0060, ISO/IEC JTC 1, Information Technology, Subcommittee SC 22, Programming Language C++, 2003.

[4] T. Binder, A. Hössinger, and S. Selberherr. Rigorous Integration of Semiconductor Process and Device Simulators. *IEEE Trans.Comp.-Aided Design of Int. Circ. and Systems*, 22(9):1204–1214, 2003.

[5] Boost. *Boost Lambda Library*. `http://www.boost.org`.

[6] Boost. *Boost Phoenix*. `http://spirit.sourceforge.net/`.

[7] Boost. *Stream - Sustainable Memory Bandwidth in High Performance Computers*. `http://www.cs.virginia.edu/stream/`.

[8] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, and J. Willcock. A comparative study of language support for generic programming. In *Proc. of the 18th Annual ACM SIGPLAN*, pages 115–134, New York, NY, USA, 2003. ACM Press.

[9] Douglas Gregor, Jaakko Järvi, Mayuresh Kulkarni, Andrew Lumsdaine, David Musser, and Sibylle Schupp. Generic Programming and High-Performance Libraries. *Int. J. of Parallel Prog.*, 33(2), June 2005.

[10] S. Halama, Ch. Pichler, G. Rieger, G. Schrom, T. Simlinger, and S. Selberherr. VISTA — User Interface, Task Level, and Tool Integration. *IEEE J.Techn. Comp. Aided Design*, 14(10):1208–1222, 1995.

[11] R. Heinzl, M. Spevak, P. Schwaha, and T. Grasser. A High Performance Generic Scientific Simulation Environment. In *Proc. of the PARA Conf.*, Umea, Sweden, June 2006.

[12] R. Heinzl, M. Spevak, P. Schwaha, and T. Grasser. Concepts for High Performance Generic Scientific Computing. In *Proc. of the 5th MATHMOD*, volume 1, Vienna, Austria, February 2006.

[13] J. J. Holdsworth. Graph Traversal and Graph Transformation. *Theor. Comput. Sci.*, 321(2-3):215–231, 2004.

[14] IμE. *MINIMOS-NT 2.1 User's Guide*. Institut für Mikroelektronik, Technische Universität Wien, Austria, 2004. http://www.iue.tuwien.ac.at/software/minimos-nt.

[15] L. Lee and A. Lumsdaine. Generic Programming for High Performance Scientific Applications. In *JGI '02: Proc. of the 2002 joint ACM-ISCOPE Conf. on Java Grande*, pages 112–121, New York, NY, USA, 2002. ACM Press.

[16] P. A. Markowich, C.A. Ringhofer, and C. Schmeiser. *Semiconductor Equations*. Springer, Wien-New York, 1990.

[17] B. McNamara and Y. Smaragdakis. Functional Programming in C++ using the FC++ Library. *SIGPLAN*, 36(4):25–30, April 2001.

[18] C. E. Oancea and S. M. Watt. Parametric Polymorphism for Software Component Architectures. In *Proc. of the OOPSLA Conf.*, pages 147–166, New York, NY, USA, 2005. ACM Press.

[19] Rainer Sabelka and Siegfried Selberherr. A Finite Element Simulator for Three-Dimensional Analysis of Interconnect Structures. *Microelectronics Journal*, 32(2):163–171, 2001.

[20] P. Schwaha, R. Heinzl, M. Spevak, and T. Grasser. Advanced Equation Processing for TCAD. In *Proc. of the PARA Conf.*, Umea, Sweden, June 2006.

[21] S. Selberherr. *Analysis and Simulation of Semiconductor Devices*. Springer, Wien–New York, 1984.

[22] J. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[23] J. Siek and A. Lumsdaine. Mayfly A Pattern for Lightweight Generic Interfaces, 1999.

[24] J. G. Siek and A. Lumsdaine. The Matrix Template Library: A Unifying Framework for Numerical Linear Algebra. In *ECOOP Workshop*, pages 466–467, 1998.

[25] M. Spevak, R. Heinzl, P. Schwaha, and T. Grasser. A Computational Framework for Topological Operations. In *Proc. of the PARA Conf.*, Umea, Sweden, June 2006.

[26] T. L. Veldhuizen and D. Gannon. Active Libraries: Rethinking the Roles of Compilers and Libraries. In *Proc. of the SIAM Workshop on Obj.-Oriented Methods for Inter-Operable Sci. and Eng. Comp. (OO'98)*. SIAM-Verlag, 1998.

[27] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. In *9th SIAM Conf. on Parallel Proc. for Sci. Comp.*, 1999. CD-ROM Proceedings.