Performance Analysis of Generics in Scientific Computing

Laurentiu Dragan Stephen M. Watt
Ontario Research Centre for Computer Algebra
University of Western Ontario
London, Ontario, Canada N6A 5B7
{ldragan,watt}@orcca.on.ca

Abstract

This paper studies the performance of generics, or templates as they are sometimes called, for scientific computing in various programming languages. In order to understand the cost of using generics, we develop a test suite for generics based on a standard numeric benchmark. We compare the results of this new benchmark for generics in C++, C# and Java, both between language implementations and against the specialized, non-generic benchmark. We also compare the efficiency of C++ with Aldor, a language originally for computer algebra relying entirely on generics. We find that the implementation of generics in current compilers must be improved before they are used for efficiency-critical scientific applications, and we identify specific areas for potential optimization.

1. Introduction

Previous studies have shown that modern implementations of C++, C# and Java now have sufficient performance for traditional scientific computing. As scientific computing evolves to take more advantage of modern programming language features, we must understand which of these can be implemented sufficiently efficiently for numerically intensive codes. Parametric polymorphism, allowing programs to be written using type parameters, is now supported by several modern programming languages. In Ada programs using parametric polymorphism are known as "generics" and in C++ they are known as "templates". This paper examines the cost of using this language feature in scientific computing. We shall use the terms "generics" and "templates" interchangeably, unless there is a particular reason to do otherwise.

Scientific algorithms are very well suited to generic style of programming due to their rich mathematical structure. The feasibility of generic code for scientific computing has been investigated in [2], [15]. Good examples of the usefulness of the generic libraries are provided by the C++ standard template library (STL) and the Boost libraries [3]. There are many computer algebra libraries using parametric polymorphism: the NTL library for number theory [4], the LinBox library for symbolic linear algebra [5], the Sum-it library for differential operators [6] and the Triade library for triangular sets [7], to restrict ourselves to just a few.

Although parametric polymorphism has been widely accepted in the symbolic computation community, it has not yet been widely adopted for numerically intensive computation. One of the pre-requisites for its adoption in this context is a clear understanding of its cost.

To address this question we have developed a benchmark for generics in scientific computing. We call this the "SciGMark" benchmark, because it is an extension of the well-known SciMark benchmark [8] using generics. The benchmark suite contains various language implementations of a number of tests, described in this paper. The first version of SciGMark contains the problems from SciMark for Java, C++, C# and Aldor in both specialized and generic form. We have added polynomial multiplication since this is representative of another typical scientific computation.

Parametric polymorphism admits many optimizations. It should be possible, in principle, to see similar performance for code that uses parametric types and hand-specialized code, provided the compiler is able to perform suitable code transformations. In their current state, however, the compilers we tested fall far short of achieving this.

We believe that providing a benchmark for generics in scientific computing can help improve this situation: Initially, Java's performance was unacceptable for numerically intensive computation. To encourage performance improvement of Java for numerical computation, benchmarks where created to measure the performance relative to higher-performance languages such as Fortran and C. We then saw a dramatic increase in performance to the point that Java implementations can be comparable to C and, in some cases, even faster. We anticipate that the same performance evolution could occur for generics.

With parametric polymorphism available in certain mainstream languages, such as C++, Java and soon C#, we foresee an increased reliance on generic code. To support this, compilers must be able to optimize generic code to an acceptable level. We therefore need benchmarks to measure the performance of compilers in this area. We hope that the SciGMark benchmark will help in this regard.

The nature of scientific computing places different emphasis on the performance of generics than other programming styles. In most object-oriented programming, objects are created and then their state is modified through the invocation of a series of methods. In mathematical computing, expressions tend to be more functional, with objects being short-lived and never modified. The optimization of generic code must take this into account.

The contributions of this paper are:

- The presentation of a benchmark suite for generics in scientific computing.
- A preliminary assessment of generics for scientific computing in C++, C# and Java.
- Observations on how the compilation of generics for scientific computing can be improved, based on our experience with the Aldor compiler.

The remainder of this paper is organized as follows: Section 2 describes aspects of parametric polymorphism in various programming languages of interest. Section 3 describes how we have composed the SciGMark benchmark suite, including the strategy used to generalize the SciMark problems. Section 4 examines certain languages in detail and describes how our generic benchmarks are applied. Section 5 presents benchmark results, for both the specialized and generalized versions of the tests. Section 6 describes potential optimizations to improve the implementation of generics, and Section 7 presents our conclusions.

2. Parametric Polymorphism in Different Languages

Parametric polymorphism has been available in experimental programming languages for some time, and is now becoming important in mainstream languages. For example, it is supported in ADA, C++, Java and Modula-3. The next release of .NET platform will also have support for generic programming, e.g. in C#.

Parametric polymorphism can be implemented in different ways, and these carry with them different overhead trade-offs. Currently, there are two main approaches: the "homogeneous" approach and the "heterogeneous" approach.

The *heterogeneous* approach constructs a special class for each different use of type parameters. For example, with `vector` from the C++ STL, one can construct `vector<int>` and `vector<double>`. Because C++ uses the heterogeneous approach, two distinct classes are generated for the above cases: one with the type parameter replaced by `int` and one with it replaced by `double`. These duplicate the source code of the `vector` generic class and produce different specialized compiled forms.

This approach has the benefit that the compiled code is specialized, and therefore fast. The drawback is that object code can be bulky, with many different versions of each class. This can cause problems due to space constraints at any level of the memory hierarchy. This approach requires much more sophisticated system software if generics are to be instantiated at run time.

The *homogeneous* approach uses the same generic class for every instance of the type parameters. Specialized behavior is achieved through function or method calls. Java uses this approach by "erasing" type information and using the `Object` class instead of the specialized form, and by casting back to target class whenever necessary. This method has overhead comparable to that of subclassing, and the code size is comparable to the non-generic version. For example, `Vector<Integer>` will be transformed to a `Vector` that contains `Object` values, and the compiler will check if an `Integer` class object is used when referring to elements of the vector. This allows the same code to be used for `Vector<Double>`.

In languages that offer bounded polymorphism, such as C# and Java, it is possible to specify an interface for that specifies the operations that are allowed on the type parameter.

3. Generalizing A Numeric Benchmark

Our first goal was to assemble a set of tests for generics that would be significant for scientific computing. It is clear that these tests should have the properties (1) that when the generics are specialized, the resulting codes give representative numeric computations, and (2) that parameterization is used in a manner we can expect in scientific computing. We have achieved these ends by basing our work on a standard test suite, introducing generic type parameters for certain numeric types. When the generics are specialized with certain parameters, the function of the original tests is recovered.

A well-recognized test for the numeric performance of Java is SciMark [8]. It was originally developed to test the performance of Java for scientific computing and has subsequently been used to analyze the performance of C# [9] [10].

We have chosen SciMark as the non-generic suite to serve as the starting point for our generic benchmark set. SciMark measures the following computational kernels for floating point performance:

Fast Fourier transform (FFT) performs a one-dimensional forward transform of 4K complex numbers. This kernel exercises complex arithmetic, shuffling, non-constant memory references and trigonometric functions.

Jacobi successive over-relaxation (SOR) on a 100x100 grid exercises typical access patterns in finite difference applications, for example, solving Laplace's equation in 2D with Dirichlet boundary conditions. The algorithm exercises basic "grid averaging" memory patterns.

Monte Carlo integration approximates the value of π by computing the integral of the quarter circle $y = \sqrt{1 - x^2}$ on $[0, 1]$. It chooses random points within the unit square and computes the ratio of those within the circle. The algorithm exercises random-number generators, synchronized function calls, and function inlining.

Sparse matrix multiply uses an unstructured sparse matrix stored in compressed-row format with a prescribed sparsity structure. This kernel exercises indirection addressing and non-regular memory references.

Dense LU matrix factorization computes the LU factorization of a dense 100x100 matrix using partial pivoting. This exercises linear algebra kernels (BLAS) and dense matrix operations.

All the tests from SciMark were reimplemented in SciGMark, but with certain numeric types replaced by generic parameters. That is, the algorithms were re-phrased to use generic type parameters, e.g. R instead of `double`.

The `double` basic type was replaced with the `DoubleRing` class. This class uses a `double` as its representation, but any other representation could be used, provided that the necessary interface is implemented. The interface requires the usual field operations $(+, -, \times, /)$, as well as certain other functions. Instantiating a generic test with `DoubleRing` for R tests the compiler's ability to optimize generic instantiation as well as to inline operations from the parameter class.

Likewise, the explicit use of complex numbers was replaced with with a generic class, `Complex<R>`, accepting a parameter type for its real and imaginary parts.

In some cases the use of complex numbers is implicit in SciMark, as is the practice in much scientific computation. For example, the FFT test uses an array of $2n$ `doubles` to represent n complex numbers. In our generic test, we replaced this use with the `Complex<DoubleRing>` class. This explicit form is more likely to be used in practice with generic codes.

To generalize successive over-relaxation, Monte Carlo, sparse matrix multiplication and LU factorization, it was only necessary to replace the `double` primitive type with the class `DoubleRing` and use the operations through the `IRing` interface. In the future versions, we may additionally use generic versions of `Matrix` and `Vector`.

Since generic polynomial arithmetic uses type parameters in another representative manner, we decided to include polynomial multiplication as a test in SciGMark. To make comparisons, it was necessary to provide both specialized and generic implementations. Both versions used a dense polynomial representation with coefficients from a prime field. The specialized version used an array of integers and performed the modular operations inline. The generic polynomial multiplication test used the generic class `DensePolynomial<R>`. The parameter R was instantiated with the `SmallPrimeField` class, which would perform the modular calculations within its arithmetic methods. This test made significant use of memory allocation to create the polynomial objects.

4. Languages

This section makes a few observations on the implementation of generics in the languages we tested, and on how this affected the implementation of the tests.

4.1. Java

The Sun compiler for Java 5 uses the GJ [11] approach to generics. The advantage of this approach is that it does not require changes to the virtual machine and has no additional performance penalty beyond the subclassing polymorphism commonly used in Java. However, by keeping the virtual machine intact, some functionality must be sacrificed.

Generic programming in Java is essentially an automatic cast on top of polymorphism implemented by a super-class (in this case `Object`). For example, a `List<Integer>` is actually transformed by the compiler to `List` that contains `Object`-type elements. (This is what is meant by type "erasure.") Elements are cast down to `Integer` when extracted from the `List`. This homogeneous approach uses the same representation for all instantiations, and gives a small code size.

One problem using generics in Java is that one cannot instantiate parameters using primitive types, such as `int`, `float`, `char` and so on. The problem is partially addressed using wrapper classes and "auto-boxing", a language feature for automatic conversion. Another problem is that type information is lost at runtime, making it impossible to construct a new instance of the type parameter. There are two ways to solve this problem: by creating a factory object to produce new instances or by using reflective features of the language, such as `Class.newInstance`.

For our benchmark, the basic form of declaration for the interfaces can be illustrated by the following:

```
interface IRing <T> {
    T a(T other_elem);
    T newInstance();
}
interface IComplex<C, E extends IRing<E>>
        extends IRing<C> {
    create(E re, E im);
    E re();
    E im();
}
public class Complex <R extends IRing<R>>
        implements IComplex<Complex<R>,R> {
}
```

Java can use bounded polymorphism by specifying the class or interface that is a supertype of the actual parameter. When the instance of the type is erased, it is erased to the given supertype, giving the compiler information about what methods can be called on the parameter `s`. This way it is possible to compile the class without knowing the actual instantiation that will be used.

By explicitly specifying the `Complex` class as a type parameter to `IComplex`, the Java type inference will issue fewer warnings and the code will be cleaner, because `IComplex` interface declare the "`a`" method as `Complex a(Complex a)`. If `IRing` and `IComplex` would not have known the `Complex` type, the method to be overridden would have been `IRing a(IRing a)`. This second case can be made to work, but with many unchecked warnings.

To work around Java shortcomings, we specified a `newInstance` method to create new values of the parameter type. Each program had to store a sample value belonging to the parameter type and use it to construct new values later on. We could have used reflective features, but reflective features are slow, and require the program to store the parameter type itself, rather than an instance value.

4.2. C++

In C++, parametric polymorphism is provided by templates. The templates of C++ are implemented using a macro engine that expands templates to generate the specialized code. When the parameter is instantiated, the compiler checks the resulted code. This approach of C++ allows more straightforward code specialization and compile-time optimization.

C++ does not support bounded polymorphism, and as a consequence compilers cannot check parametric code for correctness until it is instantiated. Type checking is thus deferred to the moment of instantiation. Whenever a template is instantiated, the actual definition of the template is specialized by replacing the template parameter with the actual instance. For example, if we want to create a container `vector` with elements of type `int`, we would write: `vector<int>`. At this point, the code for `vector` is duplicated, and the parameter of the `vector` template is replaced by `int`. The specialized code is then checked by the compiler.

In C++, the use of stack allocated objects is much faster than the heap allocated objects. Code similar to typical Java, where each object is heap allocated, will sometimes result in worse performance for C++ than Java. We therefore used stack objects for the C++ SciGMark tests.

Based on some simple tests that show that the performance of `std::vector` is close to primitive arrays and because the C++ standard template library is well supported by most of the C++ compilers, we decided to use the collections available from C++ instead of basic arrays provided by plain C.

Another interesting optimization problem was the use of another file for the `Double` class. If `Double` was defined in a separate file, the C++ optimizer did not inline the code of `Double` into the caller. So we had to either put the definition together with the declaration in the header file, or to define the `Double` class in the same file as the caller code.

4.3. C#

Although the current version of the .NET platform does not support parametric polymorphism, the next version will and beta versions are presently available.

The implementation of generics for .NET is described by Kennedy and Syme [1]. The advantage of .NET implementation of generics is that type information is retained at runtime, making possible optimizations by a just-in-time compiler, and avoiding some of the restrictions imposed by generics as implemented in Java.

C# allows use of stack allocated structures to improve performance. It also allows the use of basic types as type parameters. As opposed to Java, in C# it is possible to use `Complex<double>` instead of `Complex<DoubleRing>`. .NET uses a mixed approach by implementing the heterogeneous parametric polymorphism for the basic types (similar to C++) and a homogeneous approach for reference types (similar to Java).

Any structure that is stored in a collection is automatically boxed by the compiler leading to a decrease of performance compared to use of classes. Since our test code stores the data in arrays, using structures instead of classes would decrease the performance due to this automatic boxing/unboxing.

For the C# benchmark we used similar code as for Java, and similar generic constructions:

```
public interface IRing<T> {
    T a(T other_elem);
    T newInstance();
}
public interface IComplex<C, E>: IRing<C>
       where E: IRing<E> {
    C create(E re, E im);
    E getRe();
    E getIm();
}
public class Complex <R>:
    IComplex<Complex<R>,R> where R:IRing<R>{
}
```

4.4. Aldor

Aldor [12], [13] was designed as extension language for the Axiom computer algebra system. Later, it developed as a general purpose programming language that placed an emphasis on the uniform handling functions and types, and less emphasis on a particular object model. In Aldor, functions and types are first-class values, allowing rich relationships between mathematical structures. The type system in Aldor is organized on two levels: domains and categories, with categories representing the types of domains.

A novel feature of Aldor is its pervasive use of *dependent types*. This allows the *type* of one subexpression to depend on the *value* of another. It also allows normal functions to provide parametric polymorphism, e.g.

```
suml(R: ArithmeticType, l: List R): R == {
    s: R := 0;
    for x in l repeat s := s + x;
    s
}
```

In this example, `l` is a list that contains elements of type `R`. It is also used to specify the return type of the function.

The following example shows how to construct the generic domain `Complex` and `IRing` and `IComplex` categories. This time, `IRing` does not require a parameter because % is expanded to the proper type in `Complex`.

```
define IRing: Category == with {
    +        : (%, %) -> %;
    newArray : int -> Array %;
};

define IMyComplex(E: IRing): Category ==
IRing with {
    create: (E,E) -> %;
    getRe : % -> E;
};

define MyComplex(E: IRing): IMyComplex(E) ==
add { ... };
```

5. Results

5.1. General Results

Table 1 presents the performance results obtained for the SciGMark benchmark in the selected programming languages. The tests are abbreviated as FFT (fast Fourier transform), SOR (successive over-relaxation), MC (Monte Carlo), MM (sparse matrix multiply), LU (LU matrix factorization), PM (dense polynomial multiplication). The table shows the performance both for instantiated generic code, and hand-written specialized code.

The entries in the table are given in MFlops. The tests were performed on a Pentium IV 3.2 GHz with 1MB of cache and 2GB RAM. The operating system was Windows XP SP2. The compilers used were: Cygwin/gcc 3.4.4 for C++, Sun Java JDK 1.5.0_04 for Java, Microsoft.NET v2.0.50215 for C#, and version 1.0.2 for Aldor.

It should be noted that the *generic* version of the tests are typically an order of magnitude slower than the specialized versions. This means that the compilers cannot optimize the generic code to produce efficient code even if the code is essentially the same except for data type representation.

The specialized version of Java has similar speed to C++ and the generic code is close. It is not possible to replace `Complex<DoubleRing>` with `Complex<double>` to optimize the code, because of the way generics are implemented in Java. The performance is much better with basic type `double`, but `double` cannot be used in collections like `Vector`; it can only be used in arrays.

C# still requires some improvements to its just in time compiler. The version used for testing here is not the final product and the final version may show better results.

The data used for these tests is very small to fit into the cache. A large version of the benchmark can be run from source code, the most significant difference is seen in FFT and LU tests that run at about half performance. Due to space constraints, the results for the large data size are not presented here.

5.2. Aldor Results

The specialized version of Aldor used only the basic data types. The resulting code has performance close to C++. Unfortunately, the generic version of Aldor does not produce similar results.

Two problems were identified as the source of the weaker performance of Aldor. Due to the lazy nature of the Aldor runtime, the environments of the closures must be initialized before use. Each use of a closure verifies that the environment is properly initialized, and these checks decrease the performance. The second performance problem for the generics in Aldor is the frequent use of memory allocation.

Test	C++		Java		C#		Aldor		Size
	Gen	Spe	Gen	Spe	Gen	Spe	Gen	Spe	
FFT	59	365	23	321	7	242	1	340	1024
SOR	71	419	66	681	22	417	15	417	100x100
MC	46	65	22	26	28	62	90	203	N/A
MM	87	739	111	410	39	477	4	485	1000×5000
LU	103	780	74	982	18	403	5	553	100×100
PM	62	365	48	227	28	321	6	156	40
Composite	71	434	57	441	24	320	20	359	N/A

Table 1. Performance of the generic and specialized code in different programming languages.

Test	Aldor		C++	
	Time	Iter's	Time	Iter's
Permutations	0.43	23400	0.37	26901
Towers	0.58	17297	0.21	46924
8-Queens	0.52	19700	0.45	21987
Matrix Multiply	0.65	15386	0.20	49155
Puzzle	2.89	3484	2.16	4626
Quick sort	0.79	12538	0.66	15214
Bubble sort	0.74	13526	0.53	19089
Tree sort	1.00	10	2.00	10
FP Mat Multiply	0.69	14342	0.49	20355
Oscar FFT	0.38	26838	0.24	40719
Composite FP	1.07		1.05	
Composite int	1.43		1.29	

Table 2. Aldor *vs* C++ for Stanford benchmark

In Java, the memory allocation for objects is very cheap and in C++ the objects were stack allocated to produce similar performance to Java. In contrast, the Aldor implementation allocates all objects on the heap and relies on data structure elimination to convert these to stack references. When the compiler cannot make this transformation, the performance is significantly lower. These tests show that the Aldor optimizer is missing this opportunity in most generic settings.

Although the generic tests with Aldor do not compare well with those for C++ or Java, the specialized version used the lower-level libraries offered by Aldor and produced code with a performance similar to C++. Another example that shows that Aldor can produce reasonable performance is Stanford benchmark suite.

The original benchmark includes a recursive permutation program, a Towers of Hanoi module, code to solve the 8 queens problem, integer matrix multiplication, a compute-bound puzzle program, implementations of quick-sort, bubble sort, and tree sort, floating point matrix multiplication, and an FFT computation.

The Stanford benchmark was naïvely translated to Aldor. The modified Stanford benchmark replaced a fixed number of iterations with a variable number of iterations such that the total running time is more than 10 seconds for each test.

The results of Stanford benchmark are presented in Table 2. The times in the table are running time per iteration, in milliseconds. The benchmark was run on an Intel Pentium IV 3.2 GHz with 1MB of cache and 2GB RAM. The operating system was Linux Fedora Core 3. C was compiled using gcc version 3.4.3 20050227 (Red Hat 3.4.3-22.fc3) optimized with -O3. Aldor was compiled using version 1.0.2 and -Q5 optimization level.

6. Potential Optimizations

Our experience with Aldor has shown that procedure inlining and data structure elimination are two essential optimizations for high-performance of generics.

The results from Table 1 show a 6-18 times performance deterioration for the composite performance scores. No matter what optimization hints we gave to the compilers we tested, the performance was largely the same. In most of our cases specialization would have been possible, yet it was not performed. Additionally, some of this lack of performance is due to the use of the heap allocate objects instead of stack objects.

In each test the specialized code could be obtained from the generic code by substituting the actual parameter type, inlining its methods, and performing data structure elimination. These optimizations could be performed automatically by a compiler or a higher-level partial evaluator.

For the C++ compilers we have seen, the parameter substitution is performed but the function inlining is only sometimes done, and data structure elimination is not.

We are studying the problem of specializing domain-producing functions in Aldor [14]. The optimization performs inline expansion for functions of a domain based on the type parameters used for the instantiations. The difference between this and the macro-like expansion typically

used in C++ is that the specialization is strictly an optimization and may be partial or total.

For instance, suppose the following construction is used:

```
Vector Polynomial Complex DoubleFloat
```

This type offers operations such as addition, subtraction and multiplication. The operations in the generic type `Vector(R)` rely of exported operations of addition, subtraction and multiplication from `R`, likewise for `Polynomial(X)` and `Complex(T)`. In general, type parameters may be specified at run-time in Aldor so these generic types will rely on chains of function calls to perform arithmetic. When the parameters are all known at compile-time, such as above, it makes sense to specialize the domain by constructing a new domain, e.g. `Vector_Polynomial_Complex_DoubleFloat`. The new domain will then export *specialized* versions of its operations. As in current C++ implementations, these new operations have the advantage of knowing that some of the arguments have become constants due to type specializations. Based on this, the specialized versions of the operations can apply some aggressive optimizations to automatically produce highly optimized code.

When only some of the parameters are known at compile-time it is possible to do a partial specialization. Consider the following function that takes a type constructor `C` as an argument:

```
PolynomialVect(C: Ring) == add {
    Rep == Vector Polynomial C;
    (f: %) + (g: %): % == {
        res := new(#f);
        rf  := rep f; rg := rep g;
        for i in rf for j in rg repeat
            res(i) := i + j;
        per res
    }
}
PC == PolynomialVect(Complex DoubleFloat);
PQ == PolynomialVect(Rational);
```

In this example, a new domain constructing function would be created from the composition of `Vector` and `Polynomial`. It is possible to specialize the code of + from `PolynomialVect` by inlining the code of + from `Polynomial`. This kind of optimization produced interesting results that were presented in [14].

Experience with Aldor shows that once inlining is performed, data structure elimination is essential to obtain high performance. In mathematical programming, results are often created by one function only to be used by another. If these results are dynamically allocated, this will consume memory and trigger early garbage collection. For example the following program to compute a vector dot product creates $2n$ temporary complex numbers.

```
dot(u:Vector Complex R, v:Vector Complex R):
   Complex R ==
{
    s: Complex R := 0;
    for i in 1..n repeat s := s + u.i*v.i;
    return s;
}
```

If the arithmetic from `Complex` is inlined, we may optimize the storage allocation to obtain a program equivalent to:

```
dot(u:Vector Complex R, v:Vector Complex R):
   Complex R ==
{
    x: R := 0; y: R := 0;
    for i in 1..n repeat {
       x := x + real(u.i)*real(v.i)
              - imag(u.i)*imag(v.i);
       y := y + real(u.i)*imag(v.i)
              + imag(u.i)*real(v.i);
    }
    return complex(x,y);
}
```

The optimizations show promising results for the programming languages analyzed. However, none of the mainstream compilers tested showed support for the optimizations presented.

7. Conclusion

The importance of generics in scientific computing has already proven itself in the area of exact computation, where efficient libraries for computer algebra, number theory and sparse linear systems have been successfully deployed. We see the role of generics becoming increasingly important in numerically intensive scientific computation, if language implementations can provide adequately efficient implementations.

We see this as strongly analogous to the first efforts at scientific computation with Java: first implementations of Java had inadequate performance for typical problems in scientific computation. Programs using arrays of floating point numbers would typically run 20 times slower than C code. However with a suitable set of clear benchmarks, it has been possible to improve compiler performance for the language features required for efficient numerical computing.

We believe that a suitable set of benchmarks for generics in scientific computing will allow clear formulation of objectives for compiler improvement. To this end, we have developed the SciGMark benchmark suite, and have used it to evaluate the performance of generics in various compiler implementations.

We have compared the efficiency of generics in the Sun Java 5 Server version, the Microsoft .NET Framework/C# 2.0 beta compiler, the GCC 3.4.4 compiler for C++ and Aldor 1.0.2. We have separately compared GCC 3.4.3 with Aldor 1.0.2 on the Stanford benchmark suite.

We have seen that although generics have existed in some of these languages for some time, the implementations are still not sufficiently efficient to replace specialized code. This provides an excellent opportunity for compiler optimization, and we have discussed some ideas in this direction.

The code for our SciGMark 1.0 benchmark suite is available at: http://www.orcca.on.ca/benchmarks/scigmark/1.0

References

[1] A. Kennedy, D. Syme, *Design and Implementation of Generics for the .NET Common Language Runtime*, Proc. ACM SIGPLAN PLDI 2001, http://doi.acm.org/10.1145/378795.378797

[2] J. Gerlach, J. Kneis, *Generic Programming for Scientific Computing in C++, Java, and C#*, Springer LNCS, Vol. 2834, Sep 2003, pp 301-310.

[3] *Home Page of the Boost Project*, http://www.boost.org/

[4] V. Shoup, *NTL: A Library for Doing Number Theory*, http://www.shoup.net/ntl/doc/tour.html,

[5] J. G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner, G. Villard, *LinBox: A Generic Library for Exact Linear Algebra*, Proc. ICMS, A.M. Cohen, X.-S. Gao, N. Takayama (editors), pp. 40-50, World Scientific 2002

[6] M. Bronstein, *SUM-IT: A Strongly-Typed Embeddable Computer Algebra Library*, Proceedings of DISCO'96, Karlsruhe, Springer LNCS 1128, 1996

[7] M. Moreno Maza, *Technical Report TR 4/99, On Triangular Decompositions of Algebraic Varieties*, NAG Ltd, Oxford, UK, 1999

[8] R. Pozo, B. R. Miller, *Home Page of SciMark2 Project*, http://math.nist.gov/scimark2/

[9] W. Vogels, *Benchmarking the CLI for High Performance Computing* Software, IEE Proceedings- [see also Software Engineering, IEE Proceedings], Vol.150, Iss.5, 27 Oct. 2003, pp. 266- 274

[10] F. Gilani, *Harness the Features of C# to Power Your Scientific Computing Projects*, MSDN Magazine, http://msdn.microsoft.com/msdnmag/issues/04/03/ScientificC/

[11] G. Bracha, M. Odersky, D. Stoutamire, P. Wadler, *Making the Future Safe for the Past: Adding Genericity to the Java Programming Language*, ACM Symposium on Object Oriented Programming, 1998, http://citeseer.ist.psu.edu/bracha98making.html

[12] S. M. Watt, *Aldor*, in Handbook of Computer Algebra, J. Grabmeier, E. Kaltofen, V. Weispfenning (editors), Springer Verlag 2003, pp. 265-270.

[13] S. M. Watt, P. A. Broadbery, S. S. Dooley, P. Iglio, S. C. Morrison, J. M. Steinbach, R. S. Sutor, *Aldor User Guide*, 2000, http://www.aldor.org/

[14] L. Dragan, S. M. Watt, *Parametric Polymorphism Optimization for Deeply Nested Types*, Maple Conference 2005, Proc. of Maple conference 2005, July 17-21, 2005, Waterloo Canada, Maplesoft, pp. 243-259.

[15] T. L. Veldhuizen, M. E. Jernigan, *Will C++ Be Faster Than Fortran?*, Proc. 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97), Springer-Verlag LNCS, 1997.