Scientific Programming 11 (2003) 291–307
IOS Press

Lifting a butterfly – A component-based FFT

Sibylle Schupp
Department of Computer Science, Rensselaer Polytechnic Institute, 110 8th Street, Troy, NY 12180, USA
Tel.: +1 518 276 6887; Fax: +1 518 276 4033; E-mail: schupp@cs.rpi.edu

Abstract. While modern software engineering, with good reason, tries to establish the idea of reusability and the principles of parameterization and loosely coupled components even for the design of performance-critical software, Fast Fourier Transforms (FFTs) tend to be monolithic and of a very low degree of parameterization. The data structures to hold the input and output data, the element type of these data, the algorithm for computing the so-called twiddle factors, the storage model for a given set of twiddle factors, all are unchangeably defined in the so-called butterfly, restricting its reuse almost entirely. This paper shows a way to a component-based FFT by designing a parameterized butterfly. Based on the technique of lifting, this parameterization includes algorithmic and implementation issues without violating the complexity guarantees of an FFT. The paper demonstrates the lifting process for the Gentleman-Sande butterfly, i.e., the butterfly that underlies the large class of decimation-in-frequency (DIF) FFTs, shows the resulting components and summarizes the implementation of a component-based, generic DIF library in C^{++}.

1. Introduction

When the Fast Fourier Transform (FFT) became widely known in 1965 [3], it revolutionized the use of the Discrete Fourier Transform (DFT) because it reduced the computational complexity of the DFT to a point where the methods of the DFT analysis became applicable in practice. Since then, the DFT has grown into an important tool for many disciplines in scientific and engineering computing, including quantum physics, spectral analysis, acoustics, speech processing, optics, image processing, to name but a few. As a result, the FFT is now one of most widely used algorithms. It is therefore surprising how little this algorithm (family) has been noticed by research in software engineering. While software design has dramatically changed over the last forty years, FFT programs have changed only little, thus benefiting only little from the advances in software engineering.

One of the central ideas in software engineering today is the idea of reusability. Modern software engineering now propagates the design principles of parameterization and modularization via components [30], so that software can be adjusted at the interface level solely, with no need to change the implementation. In contrast, FFT implementations tend to be monolithic and of a very low degree of parameterization. They may allow users to customize the size and scaling factor of the transform, but exclude from parameterization quite crucial parts of the computations. Most notably the computational core, the so-called *butterfly*, "hardwires" all, or most, of its parts. The data structures to hold the input and output data, the element type of these data, the algorithm for computing the so-called twiddle factors, and the storage model for a given set of twiddle factors, all are typically unchangeable at the user's level. If, however, a butterfly would be designed in a more flexible way, be composed of loosely coupled components, an FFT could benefit in several ways:

- Users could use their own types, e.g., for the input data or the container holding them. These types could be introduced specifically for the FFT or could come from a previous computation step; they could modify the mathematical semantics of the FFT, e.g., if fixed-point arithmetic or a complex type of high precision were introduced, or provide additional functionality such as tracing or bookkeeping.
- FFT implementors could extend or replace parts of their implementation without touching the code outside the respective component. There are, for example, well over 30 algorithms for bit-reversal only [14], none of which fits all needs equally.

ISSN 1058-9244/03/$8.00 © 2003 – IOS Press. All rights reserved

While currently implementors are forced to decide on one method, a component-based approach, similar to the idea of poly-algorithms [15], would allow them to offer choices and would enable users to select the implementation that is most appropriate for them. Moreover, the maintenance costs of the program would decrease since, by construction, replacing or upgrading a component does not affect the stability of any other component.

– Simulations and experimental configurations could be run easily. It is for example often hard to compare the different computational effects of two functionally equivalent implementations without running each along with the rest of the FFT, for the interaction with other parts of the FFT can enable or hinder compiler optimizations, improve or worsen the cache behavior, and, in general, have computational effects that are hard to predict. A component-based framework allows simply swapping components in and out and ensures at the same time that all but the factor of interest is set fixed.

While the advantages of a component-based FFT usually are not questioned directly, a common concern is the potential loss of efficiency. For a component-based FFT to work practically, it is therefore of utmost importance to keep the right balance between genericity and efficiency. In other words, it is necessary to parameterize the butterfly in a way that allows not only for different instantiations but also ensures that its genericity does not violate the complexity guarantees of an FFT. How to find the right level of abstraction, is the main topic of this paper. Our approach is based on the technique of *lifting* [17,25], which, in short, starts with a non-generic butterfly and stepwise removes all, mostly implicit, assumptions that do not compromise the correctness or efficiency of the computation. The important idea thereby is to use the original, non-generic butterfly as a protection against over-generalization – as long as it can be regained by specialization, the abstraction has "lifted" the butterfly without compromising its efficiency the least.

Lifting has been the underlying technique for one of the most successful recent software libraries, the Standard Template Library (STL) [29,18]. STL became known for including genericity into highly efficient data structures and algorithms to an extent no other library had done before. Since then, STL has a tremendous impact on the design of libraries in C++ and other languages. From early successors such as MTL and POOMA [27,23] to the current, most comprehensive initiative for C++ libraries, Boost [7], the number of libraries in the spirit of STL is growing. In the area of FFTs, however, a component-based approach has been lacking yet. We started filling this gap by designing and implementing a component-based C++ library, at first for a subset of FFT algorithms: the decimation-in-frequency (DIF) FFT. The DIF library currently supports four different DIF algorithms, divided into about 120 components (3000 lines of codes).

In this paper, we focus on the core of a component-based FFT, the parameterized butterfly, show how to lift a butterfly and present the components that naturally result from the process of lifting. First, in Section 2, we recapitulate the mathematics of the FFT and the DIF and summarize the algorithmic issues of the DIF that are relevant for the design of components. Since our purpose merely is to keep the presentation self-contained we refer the interested reader to the various textbooks and surveys, e.g. [2,4,8,16,34], for a more thorough discussion. Sections 3 and 4 form the two core sections of the paper. Step by step we parameterize a monomorphic butterfly by input types, data representations, algorithms, and even storage models and give then an overview of the resulting DIF library. In the last part of the paper, Sections 5 and 6, we present experimental results and discuss related and future work. Throughout the paper we will keep the discussion mostly language-independent but illustrate the lifting process with snippets in C++. Readers need not have any deep knowledge of C++ but should be able to map the presented abstraction mechanisms to class types and other corresponding features in object-oriented or generic programming languages. Furthermore helpful is a general knowledge of languages that have polymorphic type system with type parameters, as, for example, the template system in C++, the proposed extension of the Java type system, or generics in Ada, in which the composition of components happens in an efficient, i.e., statically resolvable, way.

2. Fourier Transforms

The term *Fast Fourier Transform* denotes not just one but an entire class of algorithms: any implementation of the Discrete Fourier Transform (DFT) [10,13] with complexity $\mathcal{O}(n \cdot \log n)$ for input of length n is an FFT. There are several ways, hence several FFT algorithms, to achieve logarithmic behavior. For the most frequently used FFTs, the radix-2 FFTs for input of size 2^k, the key idea is to arrange the computation in a form

that, due to its flow graph shape, is called butterfly. Butterflies come in different forms for different radix-2 algorithms. The one we focus on is the Gentleman-Sande butterfly [11], which underlies DIF FFTs. In this section we recall the mathematical background of the DFT, the DIF, and the Gentleman-Sande butterfly and prepare the lifting process in the next section with a discussion of the relevant algorithmic issues.

2.1. The Discrete Fourier Transform

Let n be a power of 2 and denote by ω_n the following complex root of unity

$$\omega_n = e^{-2\pi \cdot j/n} = \cos(\frac{2\pi}{n}) - j \cdot \sin(\frac{2\pi}{n}) \quad (j^2 = -1).$$

The (one-dimensional) DFT defines a mapping $\mathbb{C}^n \to \mathbb{C}^n$,

$$X_k = \sum_{l=0}^{n-1} x_l \cdot \omega_n^{kl}, \quad x_l, X_k \in \mathbb{C}, \quad 0 \leqslant k \leqslant n-1, \tag{1}$$

where the sequence X_k represents n consecutive samples in the frequency domain. The number n is called the *length* of the transformation. The DFT can be represented as a system of n linear equations ($x, X \in \mathbb{C}^n$):

$$\begin{pmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ \dots \\ X_{n-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & \omega_n & \omega_n^2 & \dots & \omega_n^{n-1} \\ 1 & \omega_n^2 & \omega_n^4 & \dots & \omega_n^{2(n-1)} \\ 1 & \omega_n^3 & \omega_n^6 & \dots & \omega_n^{3(n-1)} \\ & & \dots & & \\ 1 & \omega_n^{n-1} & \omega_n^{2(n-1)} & \dots & \omega_n^{(n-1)^2} \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ \dots \\ x_{n-1} \end{pmatrix}.$$

The *inverse* or *backward* DFT can be obtained from the forward DFT by conjugating the number ω_n and scaling the summation with the inverse of the length n of the transformation:

$$x_l = \frac{1}{n} \sum_{k=0}^{n-1} X_k \cdot \omega_n^{-kl}, \quad 0 \leqslant k \leqslant n-1. \tag{2}$$

Note that in some texts the meaning of forward and backward DFT is switched and that in an implementation the scale factor could be applied to either one of them or split in two factors $\sqrt{n}$ or even entirely omitted. In either case, calculating a complex DFT of length n with the standard methods of linear algebra requires a quadratic number of (complex) multiplications.

2.2. Decimation-in-Frequency FFTs

The radix-2 DIF FFT is based on the observation that an DFT of length n can be recursively defined by two transformations of length $n/2$ and that its input can be grouped in $n/2$ butterflies, each of which performs a constant number of floating point computations. The radix-2 DIF is derived from a DFT by the following sequence of three rewrite steps. First, the summation in Eq. (1) is rewritten as the sum of two summations:

$$\begin{aligned} X_k &= \sum_{l=0}^{\frac{n}{2}-1} x_l \cdot \omega_n^{kl} + \sum_{l=\frac{n}{2}}^{n-1} x_l \cdot \omega_n^{kl} \\ &= \sum_{l=0}^{\frac{n}{2}-1} x_l \cdot \omega_n^{kl} + \sum_{l=0}^{\frac{n}{2}-1} x_{l+\frac{n}{2}} \cdot \omega_n^{k(l+\frac{n}{2})} \\ &= \sum_{l=0}^{\frac{n}{2}-1} (x_l + x_{l+\frac{n}{2}} \cdot \omega_n^{k\frac{n}{2}}) \cdot \omega_n^{kl}, \\ & 0 \leqslant k \leqslant n-1. \end{aligned} \tag{3}$$

Second, Eq. (3) is divided into an even-indexed and an odd-indexed subsequence. For even-indexed numbers, using the identities $\omega_n^{mn} = 1$ and $\omega_n^2 = \omega_{\frac{n}{2}}$, one obtains ($0 \leqslant m \leqslant \frac{n}{2} - 1$):

$$\begin{aligned} X_{2m} &= \sum_{l=0}^{\frac{n}{2}-1} (x_l + x_{l+\frac{n}{2}} \cdot \omega_n^{mn}) \cdot \omega_n^{2ml} \\ &= \sum_{l=0}^{\frac{n}{2}-1} (x_l + x_{l+\frac{n}{2}}) \cdot \omega_{\frac{n}{2}}^{ml}. \end{aligned} \tag{4}$$

In a similar way, one obtains for odd-indexed numbers:

$$\begin{aligned} X_{2m+1} &= \sum_{l=0}^{\frac{n}{2}-1} (x_l + x_{l+\frac{n}{2}} \cdot \omega_n^{(2m+1)\cdot\frac{n}{2}}) \\ &\quad \cdot \omega_n^{(2m+1)l} \\ &= \sum_{l=0}^{\frac{n}{2}-1} ((x_l - x_{l+\frac{n}{2}}) \cdot \omega_n^l) \cdot \omega_{\frac{n}{2}}^{ml}, \\ & 0 \leqslant m \leqslant \tfrac{n}{2} - 1. \end{aligned} \tag{5}$$

Setting now

$$y_l = (x_l + x_{l+\frac{n}{2}}) \quad \text{and} \quad Y_m = X_{2m}$$

in Eq. (4), yields the first sub-FFT of dimension $n/2$:

$$Y_m = \sum_{l=0}^{\frac{n}{2}-1} y_l \cdot \omega_{\frac{n}{2}}^{ml}, \quad 0 \leqslant m \leqslant \tfrac{n}{2} - 1. \tag{6}$$

Similarly, renaming

$$z_l = (x_l - x_{l+\frac{n}{2}}) \cdot \omega_n^l \text{ and } Z_m = X_{2m+1}$$

in Eq. (5), yields the second sub-FFT of dimension $n/2$:

$$Z_m = \sum_{l=0}^{\frac{n}{2}-1} z_l \cdot \omega_{\frac{n}{2}}^{ml}, \quad 0 \leqslant m \leqslant \tfrac{n}{2} - 1. \tag{7}$$

With Eqs (6) and (7), the DFT of length n has been replaced by two DFTs of length $n/2$, plus the computation of y_l and z_l, the so-called *Gentleman-Sande butterfly*. Figure 1 depicts the annotated butterfly symbol with the three arithmetic operations involved: a complex addition, a complex subtraction, and the multiplication with a corrective factor, the *twiddle factor* ω_n^l.

Since the length n is assumed to be a power of 2, $n/2$ is a power of 2 again. The same sequence of rewrite steps just described can therefore be applied to the two sub-DFTs of length $n/2$. Generally speaking, at each stage, 2^i DFTs of length 2^{n-i} can be decomposed into 2^{i+1} DFTs of length 2^{n-i-1}, at the cost of n additions and $n/2$ multiplication. Thus, the total number of operations of a DIF amounts to $n/2 \cdot log_2 n$ multiplications and $n \cdot log_2 n$ additions. Figure 2 illustrates the recursive decomposition into smaller DFTs for the case $n = 8$, reducing an 8-point DFT first to a 4-point DFT and finally to the computation of an DFT of length 2.

On a historical note, the DIF FFT goes back to the decimation-in-time (DIT) algorithm, the historically first FFT [3]. DITs and DIFs are dual to each other in the sense that a DIT can be obtained from a DIF by performing its computation and data movements in reverse order, and vice versa. Together, they constitute the two predominantly used FFTs.

2.3. DIF algorithms

As the mathematical derivation in the last subsection suggests, a DIF algorithm is a divide-and-conquer algorithm. Figure 3 lists its skeleton, a nest of three loops.

One of the interesting properties of a DIF is that it can be performed in-place. The price for such storage efficiency, however, is a permutation of either the input or the output data. As Fig. 2 shows, the repeated reordering presented in Eq. (3) results in the *bit-reversal*

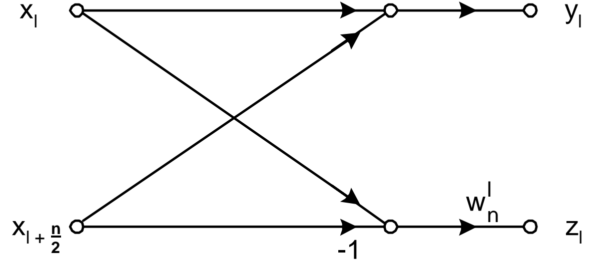


Fig. 1. Gentleman-Sande butterfly.

of the output, that is, the element on position i has been swapped with the element at position j, where i is obtained from j by writing j in binary form and reversing its bits. However, the realization of the DIF that the flow graph in Fig. 2 suggests, is not the only possible one. Bit-reversing the output can be avoided if the input is already given in bit-reversed form or if butterflies and permutations are intertwined. In the notation of Chu and George [2] we therefore distinguish the following three kinds of DIF algorithms.

- In a DIF$_{NR}$ (Fig. 2), the input is given in natural order and the output is bit-reversed. The innermost loop of the DIF, the butterfly loop (see Fig. 3), initially has length $n/2$ and decreases by a factor of 2 in each step. A disadvantage of the DIF$_{NR}$ is that the twiddle vector has to be loaded into the innermost loop.
- In a DIF$_{RN}$, the input is bit-reversed while the output is in natural order. The DIF$_{RN}$ requires an initial permutation of the input but has the advantage that the twiddle factor in the innermost loop is constant, so that the load of the twiddle container can be hoisted out. In contrast to a DIF$_{NR}$, the inner loop initially has length 1 and is increased by a factor of 2. Figure 4 shows its flow graph. Some hardware supports bit-reversed arithmetic directly.
- In a DIF$_{NN}$, both the input and the output are in natural order since the DIF$_{NN}$ performs the permutation during the computation, without an extra permutation step. A DIF$_{NN}$ needs, however, an auxiliary container, thus is the most space-expensive DIF. The DIF$_{NN}$ is sometimes called ordered DIF, its flow graph given in Fig. 4.

Our DIF library contains an algorithm, DIF$_{NRN}$, which is a DIF$_{NR}$ with subsequent bit-reversal and can thus be thought of as an in-place ordered DIF. Despite the quite different computational characteristics of these 4 algorithms, we will see that the component-based approach allows for a great amount of sharing.

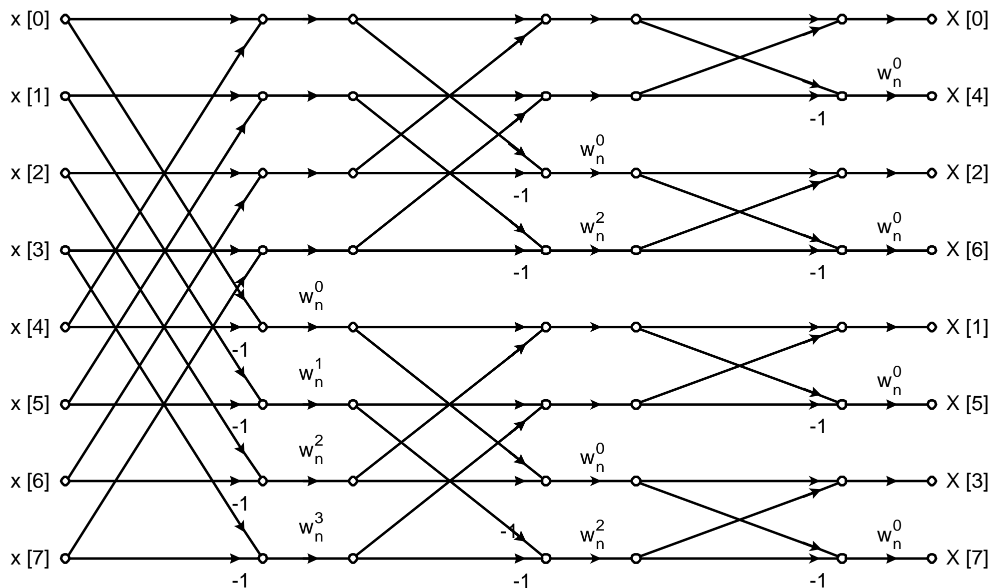


Fig. 2. The flow graph of a complete DIF decomposition of an 8-point DFT. Figure redrawn from Oppenheim & Schafer [21], Fig. 9.20.

```
for s := 1 to log(fftlength)
    divide input set into smaller blocks
    for block := 0 to (no_of_blocks - 1)
        for i := 0 to (no_of_butterflies - 1)
            get the appropriate twiddle factor and perform
            the Gentleman-Sande butterfly computation
```

Fig. 3. The control flow of a radix-2 DIF algorithm.

2.4. Twiddle factors

The second computationally intense part of a DIF, besides the loop nest just discussed, is the computation of the twiddle factors. Obviously, the trigonometric functions involved are expensive, but there are several ways to take advantage of the symmetry of complex roots of unities and other identities of the sine and cosine functions. For FFTs of length $n = 2^k$ a widely used algorithm is Singleton's algorithm [28], which introduces two auxiliary constants, $C = 1 - 2sin^2(\pi/n)$ and $S = sin(2\pi/n)$, to compute the value of $\cos((k+1)\cdot 2\pi/n)$ and $\sin((k+1)\cdot 2\pi/n)$, respectively, from the values of $\cos(k \cdot 2\pi/n)$ and $\sin(k \cdot 2\pi/n)$.

For the cache behavior of a DIF, and an FFT in general, it it also important to choose the right storage model for a twiddle set. Given a twiddle container of size 2^y, the recursive nature of a DIF implies that each step accesses only half of the twiddle factors the previous step did. If the twiddle factors are stored so that they are accessed with a power-of-2-stride, the run time of the DIF can suffer from well-known *memory bank conflicts*, a bank "busy-wait" situation that occurs when the number of banks is a power of 2 [12]. Instead of reusing a twiddle factor across several recursion steps, an alternative storage model therefore is to store multiple copies of twiddle factors and to arrange them so that the twiddle container always is traversed with a stride of 1. Obviously, the disadvantage of this storage model is that the container doubles in size.

3. Lifting the DIF

At first glance, it might seem as if the different DIF algorithms and storage models discussed in the previous section required implementations entirely disjoint from each other. The component-based approach, however, allows for a great amount of code sharing. In this

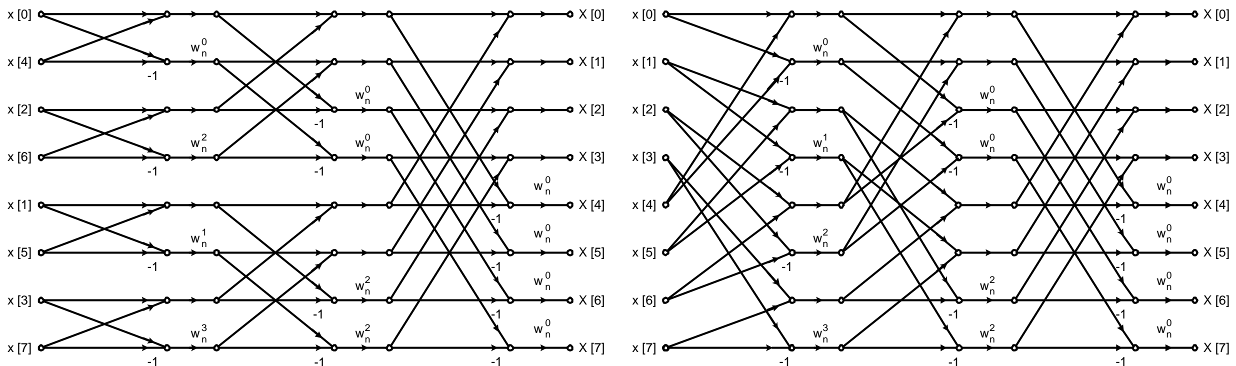


Fig. 4. The flow graph of a DIF$_{RN}$ (left) and a DIF$_{NN}$ (right) 8-point FFT. Figures redrawn from Oppenheim and Schafer [21], Figs 9.22, 9.23.

section we lift the Gentleman-Sande butterfly and the twiddle data structure and introduce the components resulting from this lifting process. To briefly reiterate what we said in the introduction, the idea behind a parameterized butterfly is to be able to obtain different implementations merely by specializing one or more of the butterfly's parameters. As we will see, five quite different kinds of abstractions are involved in its parameterization. Starting with the non-generic butterfly in Fig. 5, we remove all unnecessary restrictions, step by step and, to avoid over-generalization, in a way that always allows regaining the original code simply through appropriate parameter bindings. Although lifting itself by no means depends on C++, we illustrate the resulting components with the corresponding interfaces in our C++ implementation.

3.1. Parameterization by element types

First, we claim the "hard-wiring" of the type `complex` be an unnecessary restriction. Especially in an object-based programming style users often define their own complex data type. Instead of using the complex type that is provided by the implementation language, users might want to vary, e.g., the underlying real type, the precision, or the storage model; and for those complex types, the butterfly should work in the same way as for the built-in one. As the first step towards a generic butterfly, therefore, we lift the restriction to one particular complex type and replace the type `complex` by a type parameter. The declarations in line 1 and 3 of Fig. 5 thus become parameterized declarations and the two arrays `w` and `in` become parameterized arrays. Obviously, the original implementation can be regained by binding the new type parameter to the original type `complex`.

3.2. Parameterization by data representations

Next, we look at the arrays themselves. Figure 5 shows that altogether three kinds of operations are performed on them: they hold data, allow for random access, and allow for traversal. These three operations, by no means, are specific to the data structure of an array but characterize random access containers in general. Data types like `vector` and `deque`, for example, meet the very specification, thus should be usable as well. But, how do we add a level of indirection without performance penalty?

The Standard Template Library (STL) introduces iterators for this purpose. Operationally speaking, iterators define protocols for container traversal and element retrieval. One distinguishes 5 iterator categories, characterized by the direction, step-size, read or write access as well as the asymptotic costs of each operation. It is easy to see that these 5 categories form a partial order.

What makes iterators important in the context of components is that they serve as interfaces to containers and thereby control which types of containers to admit as actual parameters. If an algorithm is expressed in terms of iterator movements rather than operating directly on the container, this algorithm works for all containers that support the respective iterator category. At the same time, it excludes all containers with "weaker" iterators. The term "weaker" thereby refers to semantic as well as performance requirements. Since each iterator operation includes an upper bounds for its cost, which in turn depend on the underlying container, a more specialized iterator category implicitly requires a more specialized, faster underlying container, and thus restricts the set of containers an iterator can be associated with.

```
1:    complex in[], w[];
2:    int     i, twiddle, stride, length;
3:    complex W, tmp;
4:    ...
5:    W       = w[twiddle];
6:    tmp     = in[i];
7:    in[i]   = tmp + in[i + length];
8:    in[i + length] = (tmp - in[i + length]) * W;
9:    twiddle = twiddle + stride;
```

Fig. 5. A monomorphic Gentleman-Sande butterfly in pseudo-C.

For example, if, in the case of the DIF, we lifted arrays to a generic container type or even introduced a type parameter as in the previous step, we would fail to keep the efficiency promise of the DIF. It would be possible, for example, to use a list container, which is a data structure too slow for a DIF. However, if we express the generic DIF in terms of the random access iterator category [1], which requires random access in constant time, it must not be run with a list data type or other containers that are too general to define this iterator category.

As the second lifting step of the butterfly we therefore specify `operator[]` as an operation of a random access iterator instead of a container operation. The declarations in line 1 and 3 of Fig. 5 are replaced by declarations of iterator variables, the integer addition in line 9 then becomes an adjustment of the stride of the twiddle *iterator*. Figure 6 lists the interface of the function call operator that encapsulates the arithmetic operations of the Gentleman-Sande butterfly; the associated containers are defined outside the butterfly class. Decoupling algorithms and containers via iterators also offers possibilities for code reuse at an algorithmic level. The same butterfly and the same data containers, for example, can be used for both backward and forward FFTs. All one needs to do is to refine the connecting iterator from a plain random access iterator, which retrieves an element as-is, to one that performs a combined retrieval-conjugate operation.

3.3. Parameterization by scaling factors

The third abstraction we want to introduce is the abstraction from the scaling factor. As discussed in Section 2, three scaling factors are common; the monomorphic butterfly in Fig. 5 implicitly uses the scaling factor 1. To equally support all scaling factors we encapsulate each scaling operation in a function object, that is, an object that can be called as a function [1], and introduce a type parameter, say `ScaleFactor`, that can bind any of the scaling function objects. The butterfly can then be expressed in terms of an instance of the `ScaleFactor` parameter and, again without any code changes, specialized to the original, monomorphic butterfly; see Fig. 7 for the interface. To avoid the overhead of a function call thereby, it is important to use function objects instead of functions since the former ones can be inlined, at least in C++.

3.4. The lifted butterfly

Putting together the results so far, Fig. 8 lists the body of the Gentleman-Sande butterfly in its generic version. To summarize, from a monomorphic version for one particular constellation of types we have obtained a version with a three-dimensional parameter space:

$$\text{Complex Type} \times \text{Input Container} \times \text{Scaling}.$$

It is worth emphasizing that each dimension of the parameter space is conceptually unlimited – any complex type, input container, or scaling factor that meets the requirements specified can serve as component, even those that might not exist at FFT design time. In contrast, macros, the alternative to templates in C, usually require the FFT designer to determine the set of all possible choices up-front. Many C implementations can therefore offer a choice, e.g., between the types `double` and `float` for the representation of the `complex` type, but none of them permits complex numbers that are user-defined and none of them permits user-defined containers.

3.5. Parameterization by algorithms and storage models

Applying the abstractions of the previous subsection we can lift a traditional twiddle array to a parameterized twiddle container. However, the computation of the twiddle factors itself can be parameterized – two powerful abstractions are outstanding yet. The first is the abstraction from the algorithm for twiddle computations, the second the abstraction from the storage

```
class gentleman_sande {
public:
  template<typename RandomAccessIterator, typename TwiddleCoefficient>
  void operator()(const RandomAccessIterator& it,
                  const RandomAccessIterator& itU,
                  const TwiddleCoefficient&   twiddle);
};
```

Fig. 6. Parameterization by array-like data structures through decoupling and interfacing via random access iterators.

```
template<typename ScaleFactor = scale<1,identity> >
class gentleman_sande;
```

Fig. 7. Parameterization of the Gentleman-Sande computation.

```
1:  const RandomAccessIterator& it, itU;
2:  const TwiddleCoefficient&   twiddle;
3:  const ScaleFactor&          scale;

4:  typedef typename RandomAccessIterator::value_type value_type;
5:  value_type tmp(*it);
6:  *it       = scale(tmp + *itU);
7:  *itU      = scale((tmp - *itU) * twiddle);
```

Fig. 8. A generic in-place Gentleman-Sande butterfly in pseudo-C++.

model of the twiddle container. Both abstractions are more fundamental than any of the previous ones since they refer to implementation features, and they are typically not available in traditional object-oriented programming environments. For the reusability and flexibility of a system, however, it is important that users can replace implementation modules as easily as others.

STL suggests using container *adaptors*, that is, containers that are parameterized by other containers, encapsulate their specifics and thereby provide a uniform interface across different containers. Therefore, if we further lift a parameterized twiddle container to an *adaptor* to a twiddle set, which exposes access functions to the twiddle set but hides all data and functions specific for the underlying algorithm, it becomes transparent to other parts of the DIF how the twiddle set has been computed. At the same time, it becomes possible to support different twiddle computations by instantiating the adaptor in different ways.

The logic of a common interface further extends to the twiddle set and the access its adaptor provides. As discussed in Section 2.4, twiddle sets can be stored in different ways, e.g., as a single copy that has to be traversed several times or in multiple copies that have to be traversed only linearly. If we would not encapsulate their storage model, the code in the butterfly would have to take each different model into account and for example reset an iterator for one storage model, but not reset it for another. It would then be impossible to use the same butterfly for twiddle sets that are stored differently. In illustration of the two adaptors needed, Fig. 9 shows the `singleton` adaptor, which encapsulates the computation of the twiddle set, and the adaptor `twiddle_container`, which hides the twiddle set as well as its storage layout. Obviously, the `singleton` adaptor can bind `twiddle_container`'s `TwiddleSet` parameter. Together, the two adaptors yield the required transparency.

3.6. Multiple butterflies

Up to this point we implicitly assumed to deal with in-place DIFs. We have shown that the generic butterfly in Fig. 8 can be specialized to the non-generic DIF_{NR} we started with, and we could show the very same for the DIF_{RN}. For an ordered DIF, however, this genericity goes too far. Keeping two containers and using them alternately to store intermediate results, a DIF_{NN} works in fact so differently from a DIF_{NR} that we are not able to regain its monomorphic butterfly by just specializing the generic one in Fig. 8. A

```
template<typename Storage, typename TwiddleSet>
class twiddle_container;

template<int fftlength,       // length of the fft
  typename number,            // how many twiddles to compute
  typename F,                 // = identity,
  typename ComplexType,       // complex type used in arithmetics
  typename Container,         // container to store complex variables
  typename RealContainer >    // container for the real value type that
class singleton;              // is used to represent complex numbers
```

Fig. 9. Parameterization by twiddle sets and their storage models (class `twiddle_container`); various parameterizations of the twiddle computation (class `singleton`).

```
template<int stage,  // the actual stage
  typename Order,    // order of input elements (e.g, natural, reverse)
  typename Tag >     // the DIF to be performed (e.g., dif_rn, dif_nr)
class stages;
```

Fig. 10. Parameterization by different DIF computations.

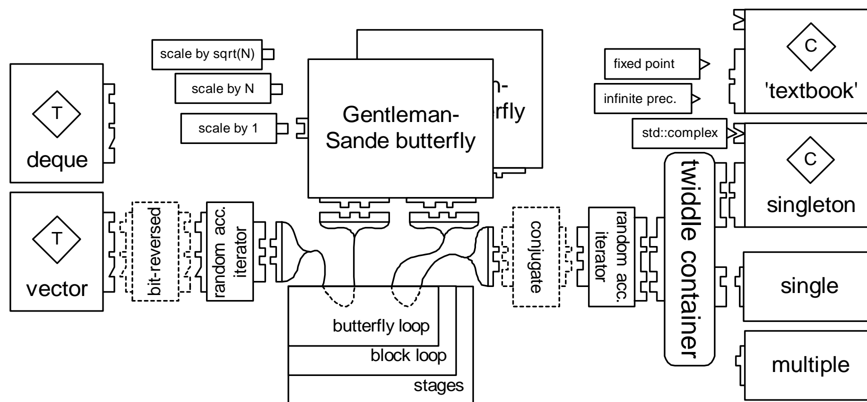


Fig. 11. The architecture of the DIF library.

second generic butterfly is needed, therefore. Starting this time with a DIF_{NN}, we have to repeat the various steps of lifting and, with arguments similar to the ones presented, determine the right level of abstraction. We skip the details here, but refer to the documentation on the project web page [24]. To provide a uniform interface we add another layer of abstraction and implement the two different algorithms as different specialization of the class template `stages` (see Fig. 10).

4. Component architecture

During the lifting process we identified five categories of components that constitute the DIF framework: algorithms, containers, iterators, function objects, and adaptors. We now give an overview of the architecture of the DIF library and its three levels: the implementation level, the library extension level, and the user's level. These three levels are disjoint so that, for example, no user ever is required to touch an implementation component or to even know about it. In the following we briefly discuss the three levels; for a full specification of each of the currently 120 components we refer to the project home page [24].

Figure 11 illustrates the four major functional units of the DIF library: the input and output data to the left of the figure, the computation and storage of the twiddle set to its right, the butterfly computation on top, and on bottom a control unit that defines the skeleton of the different DIF algorithms as sketched in Fig. 10. Organizing the majority of tasks at the implementation level, this control unit is further broken down into a set of 3 classes (see Fig. 12), which closely col-

```
template<int stage, int fftlength, int exp, int radix,
  int actuallength, class Order, class Tag, bool verbose> class stages;

template<int no_of_blocks, int blocksize, int fftlength,
  class Order, class Tag, bool verbose, int radix> class block_loop;

template<block_no,int fftlength, class Butterfly, class Tag,
  bool verbose, int radix> class butterfly_loop;
```

Fig. 12. Core components at the implementation level.

```
template<int fftlength, typename Tag>
struct fft_computation
{
    typedef fft_data_structures<Tag>::input_type input_type;
    typedef conjugate<input_type>               forward;
    typedef identity                            backward;
    typedef scale<1,identity>                   scale_factor;

    enum { unnormalized = true};
};

template<int fftlength,
  class Tag        = default_dif,
  class Direction  = class fft_computation<fftlength,Tag>::forward,
  class ScaleFactor= class fft_computation<fftlength,Tag>::scale_factor,
  bool unscaled    = fft_computation<fftlength,Tag>::unnormalized,
  bool verbose     = false,
  class Trait      = fft_trait<fftlength,Tag,verbose,Direction,
                               ScaleFactor> >
class dif_nr;
```

Fig. 13. Parameterization of the DIF_{NR} class, with default bindings to a traits class for type configurations.

laborate. In short, the class `stages` constructs one `block_loop` object per stage, which in turn constructs one `butterfly_loop` object per block, which, finally, for each butterfly in the block, "invokes" the function object that encapsulates the Gentleman-Sande computation.

A characteristic of our implementation is that, based on the techniques of meta-programming and compile-time polymorphism [6,32], the recursive division into sub-FFTs takes place at compile time. In particular, the top-level dispatcher class `stages` can "call" itself recursively at compile time by statically incrementing the template parameter that represents the current stage. As a consequence, the code of the whole computation can be laid out at compile time. Dividing a block into 2 smaller ones, determining the right input for each butterfly and all the organizational steps that contribute as constant factors to the run time of other implementations, are done at compile time in our implementation, therefore do not incur any run-time costs. The overall performance of the DIF library, thus, correlates closely to the mathematically identified FFT cost.

At the library extension level, most components are of conceptual nature. *Concepts*, one of the key innovations of STL, are specifications of the functionality of a component, its parameters, and its complexity; they are not types themselves but *models* of types. Separating the specification from an implementation, concepts allow library designers to program in a type-independent way as well as to extend a library by different implementations: every type that models a concept can serve as its realization. For example if a library designer wants to add a new algorithm for the computation of a twiddle set, s/he consults the conceptual description of the Twiddle Set and then designs the new twiddle set type so that it complies with the interface and the performance guarantees required by the concept. Core concepts of the DIF framework include Butterfly, Twiddle Set, Twiddle Container, and Bitreversal Container.

At the user's level, finally, there are two kinds of components: classes that specialize a parameter of a DIF algorithm and auxiliary types that help organizing the parameter space. The first kind of types includes twiddle set computations, storage models of twiddle contain-

```
#include "fftl/fftl.hpp"
#include <iostream>

using namespace fftl;

int main() {
    std::cout << " **** in-place natural dif_nr *** \n";
    const int fftlength = 1024;

    std::vector<std::complex<float> > v;
    // ... initialize v
    typedef singleton<fftlength>               singleton;
    typedef twiddle_container<single,singleton> twiddle_container;
    twiddle_container tc;

    // initialize iterators
    std::vector<std::complex<float> >::iterator it(v.begin());
    twiddle_container::const_iterator          t_it(tc.begin());

    // perform the FFT
    fftl::dif_nr<fftlength,fftl::default_dif>   d(it,t_it);

    fftl::container_print(v, "result in-place natural dif (dif_nr) ");
    return 0;
}
```

Fig. 14. An example main program of a DIF_{NR} computation.

Table 1
Input/output order variants

DIF algorithm	Input order	Output order
dif_nr	natural	reversed
dif_rn	reversed	natural
dif_nn	natural	natural
dif_nrn	natural	natural

ers, scale factors, butterflies, and toggles that determine the normalization, direction, and meaning of forward and backward FFT. The second kind of types mostly consists of so-called traits [19], or interface classes, a standard idiom in library design that allows users to package type configurations and to redefine them transparently to the caller's code. In illustration of the traits technique, Fig. 13 lists the trait fft_computation, which wraps up some of the implementation data types of a DIF. Because of this wrapper layer, the code of the DIF need not operate directly on any of the implementation types, but can be expressed in terms of the interface type definitions the traits class provides. As a consequence, types can be redefined without affecting the DIF code, just by changing their definition in the trait class. Figure 13 also shows how DIF algorithms rely on traits classes to establish default bindings.

In the component-based setting just presented, the responsibility of a user might seem to grow with each additional parameter. With traits, however, and, at least in C++, default parameters, the parameter space stays quite manageable. Figure 14 list an example main program in C++, where the length of the FFT is the only parameter users have to supply. Of course, users can overwrite any default parameter whenever they wish, as we have just explained.

5. Empirical results

For each of the four DIFs in our library we measured its run time, the size of its executable, and the compilation time needed. The first two measurements, run time and binary size, typically are of interest to the end user of the library, while the third kind of measurements, compilation times, matter for developers. Since there are famous examples of C++ template programs that take hours to compile, it is important to show that software development with the DIF library in fact is practical.

To summarize the main test results, dif_nr, as expected, is the algorithm with the best run-time performance. Interestingly on the other hand is that dif_nrn outperforms dif_nn, which implies that it is faster to perform an ordered DIF as a composition of DIF_{NR}

Table 2
Test platforms

	Thinkpad 600e	Origin 2000	Enterprise 450
No. of Processors	1	8	4
Processor	Celeron 366 Mhz	MIPS R10000 195 MHz	UltraSPARC-II 480 MHz
Cache	L1: 32Kb L2: 128Kb	L1: 64Kb L2: 4Mb (130 MHz)	L1: 32Kb L2: 8Mb
Memory	160Mb	1.5Gb	4Gb
Operating System	Redhat Linux 8.0	IRIX64 6.5	SunOS 5.8

Table 3
Compilers and compile flags

	Compiler	Compile flags
Thinkpad 600e	Intel C++ 7.1	-O3 -Ob2 -mp -tpp6 -mcpu=pentiumpro -march=pentiumii -ip -ipo -ansi
Origin 2000	MIPSpro 7.3.1.3m	-O3 -IPA -mips4 -Ofast -r10000
Enterprise 450	GCC 3.2	-O3 -fomit-frame-pointer -mcpu=ultrasparc -mtune=ultrasparc -finline-functions

Table 4
Sizes of executables (Kb)

	Input size = 2^3				Input size = 2^{21}			
	dif-nr	dif-rn	dif-nn	dif-nrn	dif-nr	dif-rn	dif-nn	dif-nrn
MIPS	66	76	69	69	95	105	104	98
GCC	72	78	78	72	112	118	128	112
Intel	700	720	728	712 s	792	804	932	800

with subsequent bit-reversal than to directly perform the ordered DIF_{NN}. Also as expected, the sizes of the binaries grow sub-linearly, very slowly, with the input size. Finally, the compilation times are mostly below 30 seconds even for large input sizes, thus quite acceptable. One interesting observation is that `dif_nn`, the algorithm with the slowest run time, is also the most expensive algorithm to compile. The following subsections describe the test setup and present the test results in more detail.

5.1. Test setup

In the FFT literature, it is common to measure performance in MFLOP units. MFLOPs scale the execution time by the theoretical number of floating point operations for a radix-2 FFT of length n. They are defined as follows:

$$5 \cdot n \log_2 n/(\text{ time for one FFT in } \mu s).$$

In the graphs of the performance measurements (see Section 5.2), the run time is expressed in MFLOPs. For the interpretation of the performance graphs it is important to keep in mind that higher MFLOP numbers are better than lower ones.

We tested each algorithm with 3 compilers, each on a different platform. Table 2 summarizes the characteristics of each platform, including the operating system, while Table 3 lists the compilers and compilation flags used. We varied the input size in steps of powers of 2: the smallest input size is 2^3, which is the smallest possible input size for the Singleton algorithm (see Section 2.4); the largest input size depends on the memory available, thus varies platform-dependently between 2^{21} (on a machine with 160 Mb memory) and 2^{26} (on a machine with 4 Gb memory).

We tested each input size multiple times. Since the impact of possible noise is higher for small inputs with short execution times we repeated the tests for small input size more often than the ones for larger sizes: for an input size in the range $2^3 \leqslant 2^x \leqslant 2^{10}$, we repeated each test 1000 times. For an input size between $2^{11} \leqslant 2^x \leqslant 2^{14}$, the number of repetitions was 100, for an input size between $2^{15} \leqslant 2^x \leqslant 2^{17}$ it was 30 times; for 2^{18} we repeated the tests 10 times, for the following pairs of input sizes 5 times and 3 times, respectively, and for all input sizes above 2^{22} two times.

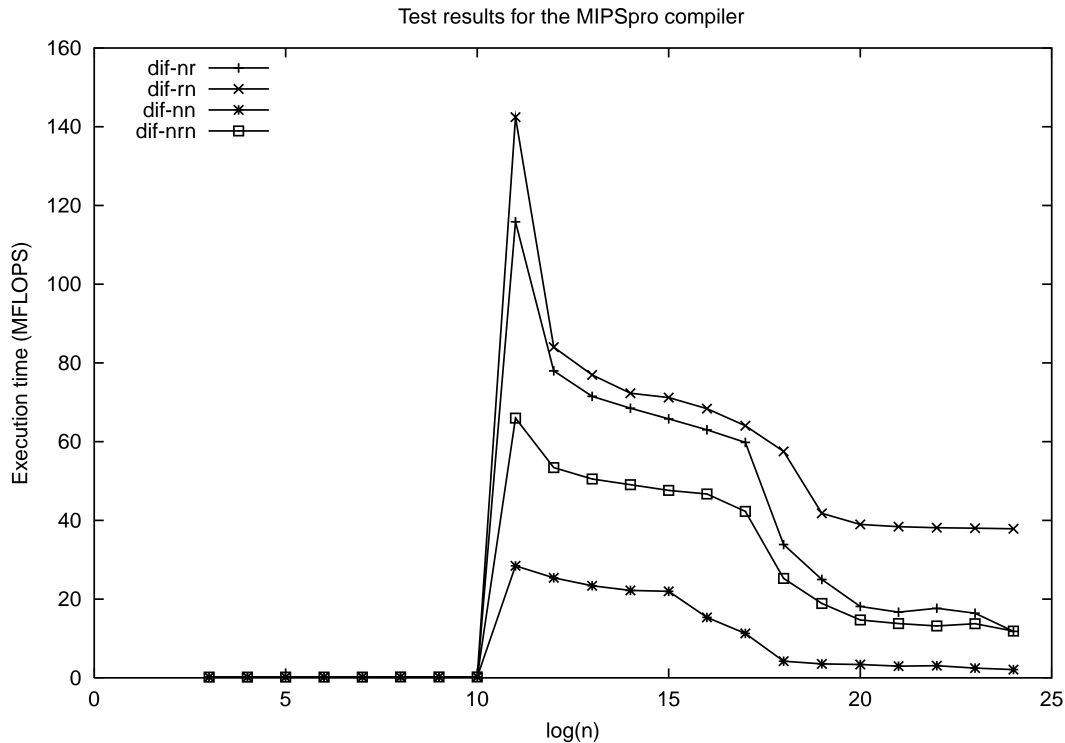


Fig. 15. Run time of 4 different DIFs, compiled with MIPSpro/MIPS.

Independently, we tested different bindings of the parameters for the DIF components. For the tests presented here, the container for the input data as well as the container for the twiddle numbers was of type `std::vector`, its element type `std::complex<float>`; the twiddle computation was done via the `fftl:singleton` class. We furthermore used the single storage model for twiddle numbers and the scaling factor 1.

The Celeron machine ran in a controlled environment and was dedicated to our tests. The other two machines, Origin 2000 and Enterprise 450, are multi-user machines with multiple processors. Since we use only one processor when running the tests, it seems safe to assume that the impact of any other processes is minimal.

5.2. Test results

We first discuss the results of the run-time tests, then the results of the compile-time tests. The sizes of the binaries are listed in Table 4.

5.2.1. Run times

The results of the run-time tests are summarized in the Figs 15–17, which plot the number of MFLOPs against the size of the input vector.

As said in the introduction of this section, the graphs confirm, at least in the MIPS and GCC/Sparc tests, that the fastest algorithm is `dif_rn`. The same is true in the Intel/Celeron test, however, only for input size $\geqslant 2^{16}$, and then, `dif_rn` is faster only by less than 1 MFLOP.

Not surprisingly, all tests agree that `dif_nn` performs worst. This algorithm, as one might recall, computes an out-of-place DIF. It requires a second container and performs alternating updates of the two containers, thus can be expected to be slower than the in-place DIFs. As we emphasized already earlier, it is almost always faster to perform an ordered DIF as a composition of DIF_{NR} with subsequent bit-reversal than to directly perform the ordered DIF_{NN}. In the MIPS and Intel/Celeron tests, the trade-off between the two out-of-place DIFs, `dif_nn` and `dif_nrn`, is at input size 2^{11}. On GCC/Sparc, we can observe the trade-off point for the input size 2^5.

The relative performance of the 4 algorithms is the same for MIPS and GCC/Sparc. For the Intel/Celeron tests, on the other hand, the run times are for the most

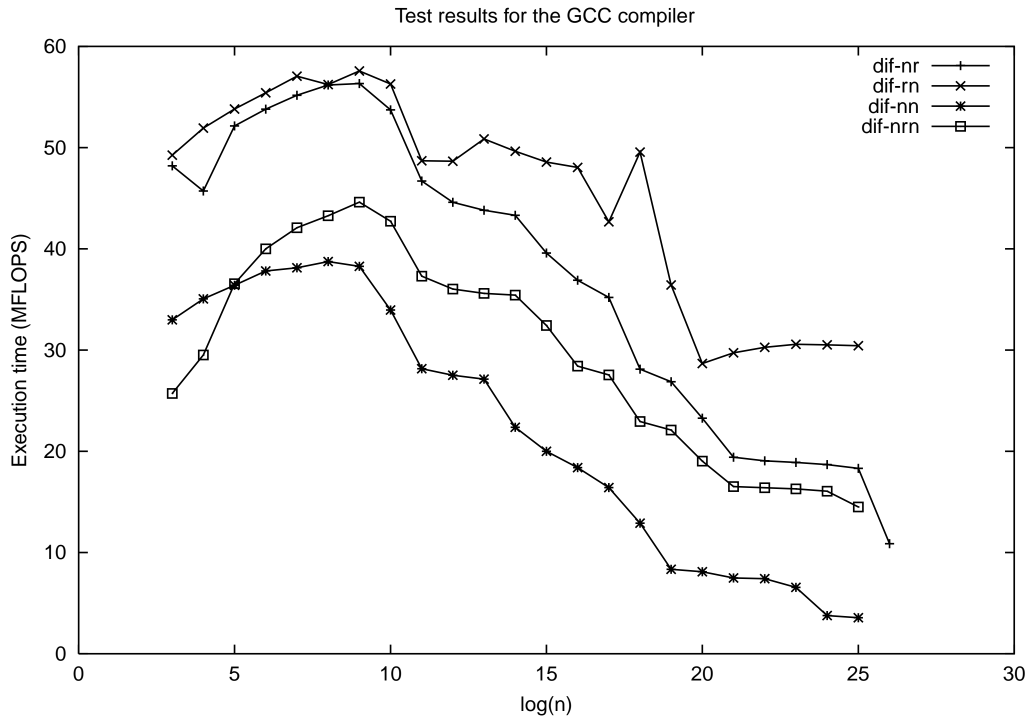


Fig. 16. Run time of 4 different DIFs, compiled with GCC/Ultra Sparc.

part too close to each other (mostly less than 1 MFLOP apart) to allow for a meaningful ranking. The overall shapes of the graphs for the 4 algorithms are similar to each other as well as across the three tests, with a peek performance around the input size of 2^{11} (for MIPS, 2^9 for GCC/Sparc, 2^{12} for Intel/Celeron).

The overall speed, finally, is the best in the MIPS tests where we reach up to 150 MFLOPS. Roughly speaking, the performance in the MIPS tests is between 2–3 times faster than in the GCC/Sparc tests and between 15 (for the peek values) to 6 (for larger values) than in the Intel/Celeron tests. Since the MIPS processor is much slower than the Sparc, it seems to be the MIPS compiler that is responsible for the better performance. On the other hand, and surprisingly, the Intel/Celeron times are significantly better than the times on the MIPS platform for small input sizes.

5.2.2. Compilation times

The results of the compilation-time tests are summarized in Fig. 18. Perhaps the most important result is the overall compilation time. As we noted already in the introduction, the compilation times do not exceed 30 seconds even for large input sizes; the only exception is `dif_nn` on GCC/Sparc. We can therefore claim that software development with the DIF library is feasible. Another positive result is the overall sub-linear behavior of the compilation time. In contrast to several template programs elsewhere that face exponential compilation times, our implementations scale in this regard.

On all machines, the algorithm `dif_nn` takes the longest time to compile. As in the case of the run-time measurements, this behavior can be expected for an out-of-place DIF, which requires extra data structures. Since one can observe that its compilation times, relative to the ones for the other algorithms, are best on the MIPS, a little worse on GCC/Sparc, and many times slower on Celeron/Intel, cache size again seems to be an important factor.

The graphs in Fig. 18 also show two unexpected results. For one, we expected the compilation times for `dif_nr` and `dif_rn` to be almost identical, since these algorithms are very similar in structure. However, this is the case only in the Celeron/Intel tests. In the MIPS and GCC/Sparc tests, the compilation times not only differ noticeably but also are separated by the measurements for `dif_nrn`, which is an algorithm of a very different structure. Second, the MIPS compiler exhibits a strange but consistent irregularity for small values. While the first phenomenon might be just coincidence, we have currently no good explanation for the second one.

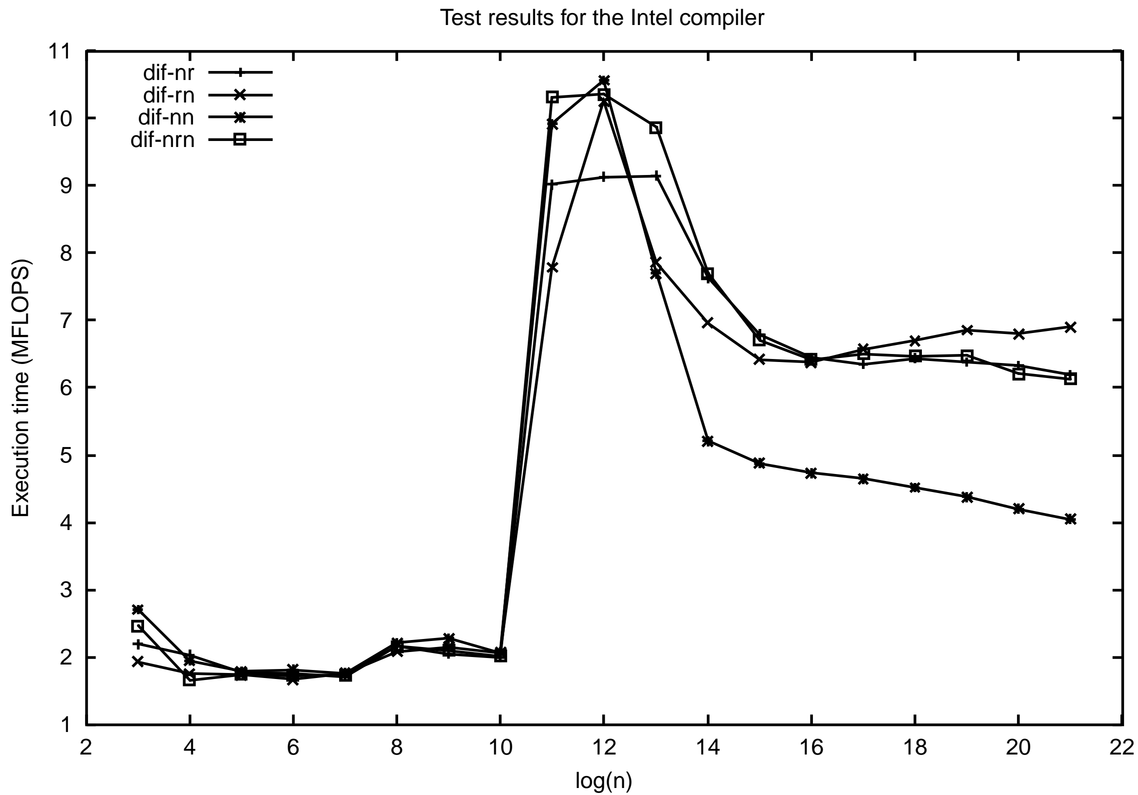


Fig. 17. Run time of 4 different DIFs, compiled with Intel/Celeron.

6. Related and future work

With its different kinds of parameterization, including the parameterization with implementation features, our DIF library is strongly influenced by STL and therefore closest to other scientific libraries in the spirit of STL, most notably Blitz++ and POOMA [31,23]. In Blitz++, however, the FFT is merely an exercise in meta-programming without much emphasis on components, while POOMA meanwhile seems to have ceased its support for an FFT. Although the POOMA user's guide describes a class template for multi-dimensional, parallel FFT, its distribution (pooma-2.2.0) lacks the respective code. The MTL [26,27], finally, was the first library we are aware of that, for matrices, made different storage types first-class components.

In a more general sense our work is related to the current trend we see in scientific programming, to move from lower programming languages such as Fortran and C to higher ones, which support classes and other mechanisms for abstraction. An important step in this direction is the Vector, Signal, and Image Processing Library (VSIPL) [33], an open industry standard that achieves portability across different memory and processor architectures through an object-based style, which enables the encapsulation and abstraction from different memory models and array representations. Version 1.1 of VSIPL is written in C but the standardization forum is currently discussing bindings to C++. In fact, the number of both open-source and commercial libraries that now provide implementations in C++ or Java, or at least interface higher languages, long has exceeded a size that would allow giving a complete reference. We therefore only point out the recent interest in the field of digital signal processing, the major application of FFTs, to develop compilers for DSP processors in C++ rather than in special-purpose languages, and these compilers rely on FFTs that are written in C++ (see, e.g. [22,5]).

For the further development of our own DFT project we identified the following three next steps. First, we want to complement the current DIF algorithms by the corresponding decimation-in-time (DIT) ones. Since DITs and DIFs are dual to each other, we expect their implementations to parallel each other. Whether after lifting and at a yet higher level of abstraction, DIFs and DITs could be further unified, is another inter-

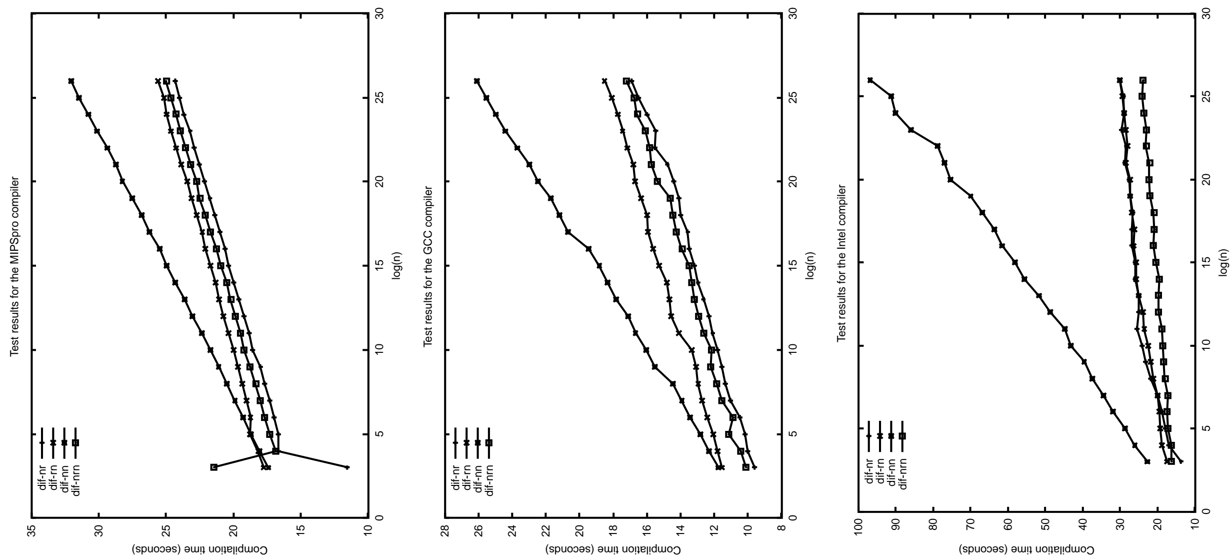


Fig. 18. Compile times in seconds for MIPSpro/MIPS (left), GCC/Ultra Sparc (middle), and Celeron/Intel (right); figures rotated by 90°.

esting topic, about which, however, we currently can speculate only. Second, we need to separately implement small-size DFTs, both for efficiency reasons and to be able to handle input of arbitrary size. As a consequence, we also need to extend our current metaprogram so that it can statically select among the different decompositions into sub-FFTs that will then be possible. Third, we need to work on optimizations of the library. We plan on applying techniques of program transformation, in particular for arithmetic operations with constant values. Program transformation allows for optimizations that are decoupled from the actual code, thus extensible for example by optimizations demanded by new user-defined types and their arithmetic. Our goal is to reach execution times that are on a par with the FFTW package ("Fastest Fourier Transform in the West"), an open source implementation that has proved to be competitive with vendor-tuned implementations [9].

7. Acknowledgments

We thank an anonymous reviewer, whose suggestions greatly helped improving the presentation of the paper. Marcin Zalewski helped with the Intel tests.

References

[1] M. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, Addison-Wesley, 1999.

[2] E. Chu and A. George, *Inside the FFT Black Box*, CRC, 2000.

[3] J.W. Cooley and J.W. Tukey, An algorithm for the machine calculation of complex Fourier series, *Math. Comp.* (1965), 297–301.

[4] T.H. Cormen, C.E. Leiserson, R.R. Rivest and C. Stein, *Introduction to Algorithms*, (2nd ed.), MIT Press, McGraw-Hill Book Company, 2001.

[5] CynApps, Cynlib, http://www.cynapps.com.

[6] K. Czarnecki and U.K. Eisenecker, *Generative Programming – Towards a New Paradigm of Software Engineering*, Addison Wesley Longman, 2000.

[7] B. Dawes and D. Abrahams, The Boost library initiative, http://www.boost.org.

[8] P.M. Embree and D. Danieli, *C++ Algorithms for Digital Signal Processing*, PTR PH, 1999.

[9] M. Frigo, *A Fast Fourier Transform Compiler*, in Proc. of the 1999 ACM SIGPLAN Conf. on Prog. Language Design and Implementation, May 1999, pp. 169–180.

[10] C.F. Gauss, *Gauss' Collected Works*, (Vol. 3), chapter Theoria interpolationis methodo nova tractata, 1886, pp. 265–303.

[11] W.M. Gentleman and G. Sande, *Fast Fourier transforms – for fun and profit*, (Vol. 29), in Proc. Joint Computer Conf., 1966, pp. 563–578.

[12] J. Guiliani and D. Robertson, *Performance Tuning for the Cray SV1*, in: Cray User Meeting, 2000, http://www.cug.org/CUG/.

[13] M.T. Heideman, D.H. Johnson and D.S. Burrus, Gauss and the history of the FFT, *IEEE Trans. on Acoustics, Speech, and Signal Processing*, **1**(4) (1984), 14–21.

[14] A.H. Karp, Bit reversal on uniprocessors, *SIAM Review* **38** (1966), 1–26.

[15] J. Li, A. Skjellum and R.D. Falgout, A poly-algorithm for parallel dense matrix multiplication on two-dimensional process grid topologies, *Concurrency: Practice and Experience* **9**(5) (1997), 345–389.

[16] D. Maslen and D. Rockmore, *Generalized FFTs – A Survey of some Recent Results*, in Proc. 1995 DIMACS Workshop in Groups and Computation, 1995.

[17] D. Musser and A. Stepanov, Algorithm-oriented Generic

Libraries, *Software-Practice and Experience* **27**(7) (July 1994), 623–642.

[18] D.R. Musser, G.J. Derge and A. Saini, *STL Tutorial and Reference Guide. C++ Programming with the Standard Template Library*, (2nd ed.), Addison Wesley, 2001.

[19] N. Myers, A new and useful template technique, in *C++ Gems*, B. Lippman, ed., Cambridge University Press, December 1996.

[20] H.J. Nussbaumer, *Fast Fourier Transforms and Convolution Algorithms*, Springer Verlag, 1982.

[21] A.V. Oppenheim and R.W. Schafer, *Discrete-Time Signal Processing*, (2nd ed.), Prentice Hall Signal Processing Series, March 1989.

[22] Open SystemC Initiative (OSCI), SystemC, http://www.systemc.org.

[23] J.V. Reynders, P.J. Hinker, J.C. Cummings, S.R. Atlas, S. Banerjee, W.F. Humphrey, S.R. Karmesin, K. Keahey, M. Srikant and M. Tholburn, POOMA: A framework for scientific simulations on parallel architectures, in: *Parallel Programming using C++*, G.V. Wilson and P. Lu, eds, MIT Press, 1996, pp. 553–594.

[24] S. Schupp, The FFT Template Library (FF⟨T⟩L), http://www.cs.rpi.edu/~schupp/entries/SOFTWARE/fftl.

[25] S. Schupp, How to lift a library, Techincal Report WSI-7-96 Fakultät für Informatik, Universität Tübingen, 1996.

[26] J.G. Siek, A modern framework for portable high performance numerical linear algebra, Master's thesis, Notre Dame, 1999.

[27] J.G. Siek and A. Lumsdaine, The Matrix Template Library: A Generic Programming Approach to High Performance Numerical Linear Algebra, in: *Internat. Symp. on Comp. in Object-Oriented Parallel Environments*, 1998.

[28] R.C. Singleton, A method for computing the fast Fourier transform with auxiliary memory and limited high-speed storage, *IEEE Trans. Audio and Electroacustics* **15** (1969), 91–98.

[29] A.A. Stepanov and M. Lee, The Standard Template Library, Technical Report HP-94-93, Hewlett-Packard, 1995.

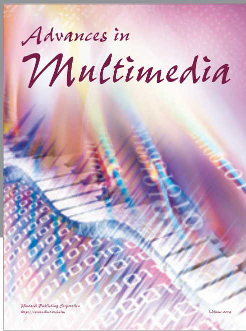
[30] C. Szyperski, *Component Software. Beyond Object-Oriented Programming*, ACM Press, 1999.

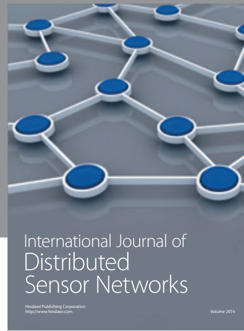
[31] T. Veldhuizen, Blitz++, http://oonumerics.org/blitz/.

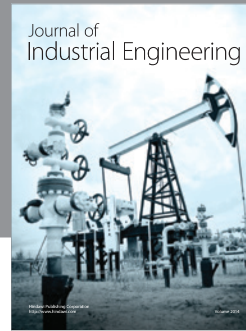
[32] T. Veldhuizen, Techniques for Scientific C++, 0.3 edition, August 1999, http://www.extreme.indiana.edu/~tveldhui/papers/techniques.

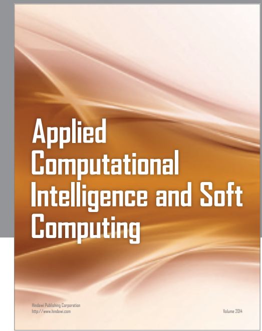
[33] VSIPL, Vector, Signal, and Image Processing Library. The Open Industry Standard for Signal Processing, http://www.vsipl.org.

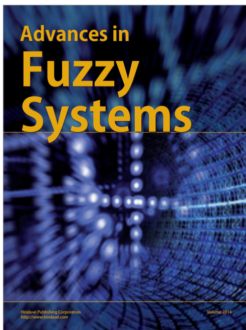
[34] K.R. Wadleigh and I.L. Crawford, *Software Optimization for High Performance Computing*, Hewlett-Packard Professional Books, 2000.











Hindawi

Submit your manuscripts at
http://www.hindawi.com