Computing and Informatics, Vol. 37, 2018, 269–290, doi: 10.4149/cai_2018_2_269

META-PROGRAMMING AND POLICY-BASED DESIGN AS A TECHNIQUE OF ARCHITECTING MODULAR AND EFFICIENT DSP ALGORITHM IMPLEMENTATIONS

Ireneusz Gawlik

Department of Electronics, AGH University of Science and Technology
Kraków, Poland
e-mail: igawlik@agh.edu.pl

Szymon Pałka, Tomasz Pędzimąż, Bartosz Ziółko

Department of Electronics, AGH University of Science and Technology
Kraków, Poland
&
Techmo, Kraków, Poland
e-mail: {pszymon, pedzimaz, bziolko}@agh.edu.pl

Abstract. Meta-programming paradigm and policy-based design are less known programming techniques in Digital Signal Processing (DSP) community, used to coding in pure C or assembly language. Major software components, like C++ STL, have proven usefulness of such paradigms in providing top performance of highly optimised native code, along with abstraction and modularity necessary in complex software projects. This paper describes composition of DSP code using these techniques, bringing as an example implementation of Feedback Delay Network (FDN) artificial reverberation algorithm. The proposed approach was proven to be practical, especially in case of prototyping computationally intense algorithms. To provide further performance insight, we discuss the techniques in context of other optimisation methods, like Single Instruction Multiple Data (SIMD) instruction sets usage and exploitation of superscalar architecture capabilities.

Keywords: C++, low level optimisations, policy-based design, template meta-programming, SIMD, FDN

1 INTRODUCTION

Adjusting signal processing algorithms to achieve desired results, while conforming to strict performance restrictions, is a challenge of many DSP designers. We have faced similar issues during our work on acoustical signal processing code, that had to satisfy both top performance requirement (real-time processing of hundreds of audio streams using consumer grade hardware) and high level of subjective sound quality [11].

Most of the high performance DSP programming is done in assembly or C language (especially in the embedded/low-energy systems). Along with project size, using procedural paradigm leads to unmanageable code, which lacks composability and is error prone. While code complexity problem is mostly addressed by composing program structure in object oriented manner (using languages like C++, C#, Java, etc.), such solution may lead to severe performance impact, particularly in case of abstracting small, loosely-coupled code blocks. In some cases of DSP code, it means only few arithmetic operations per function call. Similarly, architecting modular code in C language involves function pointer management, that, in fact, is similar to what happens behind the scenes during virtual method calls in the object oriented languages like C++ or Java. Negative performance impact of virtual method calls is directly connected to the modern pipelined CPU architectures [12], memory access bottleneck [15] and function call overhead [6] (that cannot be *inlined* by compiler).

Common approaches to DSP programming may be divided into following classes:

- Assembly programming, usually targeted for dedicated DSP processors.
- C/C++ programming, with the use of popular DSP libraries [8, 27].
- Dedicated, usually functional domain specific languages (DSL), such as Faust [19], SuperCollider [16], ChucK [33, 32] or Kronos [18]. These languages are usually targeted for audio processing and sound synthesis.

As domain specific languages are not targeted for general purpose DSP programming, and assembly coding is error prone, nonportable, and unsuitable for complex problems, we focus on comparing our approach to the traditional C/C++ programming with use of optimised DSP libraries.

High performance DSP libraries, like Intel IPP [27] or FFTW [8] are widely available. Most of them provide implementations for some of the frequently used DSP building blocks, like discrete transforms, Finite Impulse Response (FIR) and Infinite Impulse Response (IIR) filtering, etc. There are also a specialised solutions, developed to automatically generate highly optimised machine code for many integral transforms [22]. While such primitives cover plethora of applications, not all problems may be solved efficiently using these functionalities, as shown in the following sections.

In this paper, we propose an approach to solve the problem of contradictory requirements, namely code *performance* and *composability*. Both of these requirements can be efficiently satisfied using template based meta-programming, implemented in C++ language. We also address *readability*, which is often a concern in the case of template meta-programming. This work is organised as follows: Section 2 provides an introduction to programming concepts used in proposed solution, namely template-metaprogramming and policy-based design. Section 3 motivates the use of proposed technique in DSP programming, describes general concepts and provides examples of simple algorithm implementations, along with considerations about SIMD instruction set usage. Section 4 describes an example of the modular implementation of a complex algorithm on the case of Feedback Delay Network reverberator. Discussed implementation has been applied in beam-tracing audio engine for computer games [35], proving its applicability in a real world multimedia problems. Section 5 reviews related work and possible further enhancements. Section 6 concludes.

Although all examples shown in this paper were tested on current x86-64 processor architectures [12], described concepts apply to any processor architecture, including specialised DSP cores and GPGPU units [5].

2 BACKGROUND

2.1 Template Metaprogramming

As C++ language evolved [28, 20], the standard committee introduced generic programming mechanisms to allow creating efficient and reusable code in fully type-safe manner. The C++ template system has been proven to be Turing-complete [31]. Therefore, any computation, that is computable in principle, may be expressed in form of templates and executed during compile time [1].

Template metaprogramming techniques have been presented in numerous publications [1, 3, 30, 26]. When used properly, they become very powerful tools for highly optimised code generation during compile time [30]. As an example, we present an implementation of fast Walsh-Hadamard transform based on meta-programming. Typically, highly optimised transform libraries like FFTW provide dedicated, assembly coded routines for small transform sizes [9]. As many fast transform algorithms express a recursive nature, they are fairly easy to implement using template meta-programming, which results in assembly closely matching optimised code written by a skilled programmer.

```
template<size_t stride,
         size_t offset,
         typename value_type,
         size_t order>
inline typename
  std::enable_if<stride != 0, void>::type
fwht(std::array<value_type, order>& vector)
{
    for (size_t i = 0; i < stride; ++i)
      butterfly(vector[offset + i],
        vector[offset + i + stride]);
    fwht<stride / 2, offset, value_type,
      order>(vector);
    fwht<stride / 2, offset + stride, value_type,
      order>(vector);
}

template<size_t stride,
         size_t offset,
         typename value_type,
         size_t order>
inline typename
  std::enable_if<stride == 0, void>::type
fwht(std::array<value_type, order>& vector)
{} // terminate template recursion
```

Listing 1. Template meta-programming based FWHT implementation

Code listing 1 presents template meta-programming based implementation of fast Walsh-Hadamard transform [2]. Algorithm is implemented in straightforward, recursive manner, using divide and conquer methodology. Such approach results in highly expressive code, that is readable despite meta-programming specific code parts. The advantage of meta-programming implementation lies in fact, that compiler is able to inline all recursive calls, because the recursion depth is known at compile time. There is no need to apply more complex, loop based solutions. The compiler is also able to unroll *for* loop inside *fwht* template instances, as the number of iterations is also defined at compile time. As this example shows, guiding the compiler to generate efficient code is as simple as providing appropriate parameters in template class instantiation. Presented technique is applicable whenever transform size can be determined during compile time (which is usually the case).

Loop unrolling may be also defined *explicitly* using meta-programming, as shown in code listing 2. Depending on compiler used, such technique may yield better results. However, its usefulness needs to be validated by measuring execution time, as latest code optimisations manuals point out that loop unrolling may lead to micro-op cache issues [7]. In such case, partial unrolling (e.g. groups of 2 or 4 consecutive iterations) may be applied.

```
template <size_t index,
          size_t offset,
          size_t stride,
          typename value_type,
          size_t order>
inline typename
```

```
  std::enable_if<index != 0, void>::type
unroll(std::array<value_type, order>& vector)
{
  unroll<index - 1, offset, stride,
    value_type, order>(vector);
  butterfly(vector[offset + index - 1],
    vector[offset + index - 1 + stride]);
}

template <size_t index,
          size_t offset,
          size_t stride,
          typename value_type,
          size_t order>
inline typename
  std::enable_if<index == 0, void>::type
unroll(std::array<value_type, order>& vector)
{} // terminate unrolling
```

Listing 2. Template meta-programming based loop unrolling

In presented implementation, coupled with explicit loop unrolling, single call of specialised *fwht* function results in assembly code free of any function calls as well as jump instructions.

Code listing 3 shows assembly for 4-point fast Walsh-Hadamard assembly code, generated by MSVC 11.0 (Microsoft Visual Studio 2012 Update 4) from discussed function templates. Such implementation may leverage superscalar capabilities through Out Of Order (OOO) execution, present in all modern processors.

```
vmovss      xmm1,dword ptr [rbx]
vaddss      xmm0,xmm1,dword ptr [rbx+8]
vmovss      dword ptr [rbx],xmm0
vsubss      xmm1,xmm1,dword ptr [rbx+8]
vmovss      dword ptr [rbx+8],xmm1
vmovss      xmm2,dword ptr [rbx+4]
vaddss      xmm0,xmm2,dword ptr [rbx+0Ch]
vmovss      dword ptr [rbx+4],xmm0
vsubss      xmm1,xmm2,dword ptr [rbx+0Ch]
vmovss      dword ptr [rbx+0Ch],xmm1
vmovss      xmm3,dword ptr [rbx]
vaddss      xmm0,xmm3,dword ptr [rbx+4]
vmovss      dword ptr [rbx],xmm0
vsubss      xmm1,xmm3,dword ptr [rbx+4]
vmovss      dword ptr [rbx+4],xmm1
vmovss      xmm2,dword ptr [rbx+8]
vaddss      xmm0,xmm2,dword ptr [rbx+0Ch]
```

```
vmovss        dword ptr [rbx+8],xmm0
vsubss        xmm1,xmm2,dword ptr [rbx+0Ch]
vmovss        dword ptr [rbx+0Ch],xmm1
```

Listing 3. Generated 4-point FWHT machine code

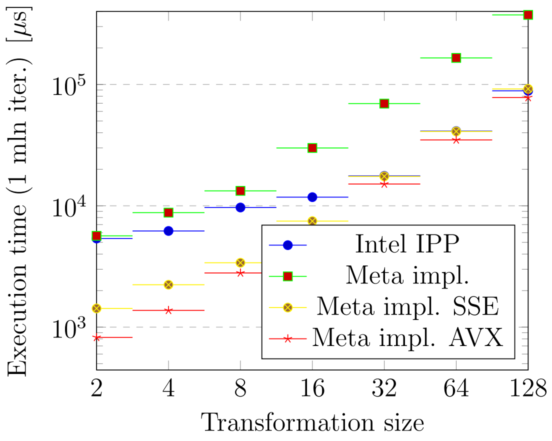


Figure 1. Comparison of fast Walsh-Hadamard transform execution time, depending on implementation used. Meta-programming based implementation is described in code listing 1. All compiler optimisations were enabled, including general optimisation (02 flag), inlining all suitable functions (Ob2 flag) and favouring fast code over code size (Ot flag). Test was performed on hardware featuring Intel Core i7-3770K 3.5 GHz processor.

As shown in Figure 1, code presented in listing 2 provides performance which is on a par with optimised Intel IPP routines in case of AVX vectorised implementation (IPP also uses AVX instruction set). In fact, proposed solution exhibits smaller execution overhead per each FWHT iteration.

2.2 Policy-Based Design

Policy-based design is a programming paradigm introduced by Andrei Alexandrescu in [3]. Its goal is to mimic the behaviour of the strategy design pattern [10], but using only static, compile time meta-programming techniques. That means that the only restriction added to strategy pattern requirements, is the need to fully determine the behaviour of the configurable component at compile time. This requirement enables policies to be an effective method of class customisation, as long as behaviour of this class may be defined at compile time.

The use of policy-based design involves providing a set of policies by passing them to the class template as template arguments. Such policies may provide static methods, type definitions or be a super-classes of the specialised class template.

Policy-based design, however, may require some experience to be implemented correctly. Most importantly, all of the policies need to be designed mutually orthogonal, so that replacing one does not affect behaviour of any of the others. Moreover, the designer may be unable to use template specialisations with different policies interchangeably (without any user code modifications), as policy-based design may involve class interface changes, i.e., when the specialised template class inherits from the policy class.

3 ENCODING DSP BLOCK DIAGRAMS USING POLICY BASED DESIGN

This section explains how metaprogramming techniques and policy-based design can be employed to aid DSP programming. We describe how to encode algorithms based on block diagrams with predefined components, using specific examples.

3.1 Motivation

Ease of experimentation with different DSP designs is the primary motivation to apply policy-based design and meta-programming methods. Because code generated using these techniques can be highly optimised during compilation phase, programmers are enabled to test both correctness and performance of the given design.

Modular design involves decomposition of the DSP diagrams into small, interchangeable components, thus embracing code reusability. As mentioned in previous sections, typical object oriented design entails virtual method calls. Such methods, when used frequently, can have negative performance impact. This is visible particularly in case of functions that contain only a few numerical operations. In contrast to classical object oriented desing, C++ template based approach allows to achieve similar level of modularity, if only polymorphic behaviour may be determined during compile time. Static polymorphism allows for code decomposition into small, individual pieces, with negligible performance impact, because compiler has much more freedom to perform code optimisations. Therefore, DSP applications, that usually struggle for top performance, benefit from the proposed approach.

3.2 General Concepts

Code composition based on DSP diagrams requires direct mapping of block elements and connections between them into template classes. Primarily, consistent naming convention needs to be enforced, as appropriate naming acts as an interface between all elements in DSP diagram. For example, in languages like C++, that allow for operator overloading, DSP blocks may be represented as *functors* (listing 4).

```
template<typename input_block>
class processing_block
```

```
{
public:
  output_type operator()(void)
  {
    output_type out = _input();
    // signal processing code
    return out;
  }
private:
  input_block _input;
};
```

Listing 4. Static variant of decorator pattern applied in DSP processing block

Each DSP block may perform on different pairs of input/output types. Therefore, heterogenic fixed-point/floating-point processing is enabled and various numerical precision representations may be used. Also, arrays of samples may be processed, if only it is necessary. Ability of combining arbitrary signal processing primitives, as long as connections between blocks match the input/output type pair, is the key feature that makes the proposed technique suitable for encoding DSP diagrams. Moreover, type-safety checks performed during compilation help to ensure correctness of built processing pipelines.

3.2.1 Sequential/Parallel DSP Blocks as Heterogenic Containers

As each DSP block needs to be represented in form of separate type, collections of blocks (connected either sequentially or in parallel) may be represented in following ways:

- Pairs of blocks, analogous to pair containers in most languages.
- Lists of blocks, implemented similarly to tuples.
- Type policy based containers, discussed later in this section.

Representing connections by pairs provides the same level of expressiveness as other solutions. Motivation to provide other ways of connecting blocks lies strictly in *ease of programming* and *readability*.

All DSP block connectors need to exhibit consistent interface in order to act itself as DSP building block (e.g. need to be implemented as functors). Code listing 5 provides example implementations of pair connectors, both for serial and parallel sum connection.

```
template<class first_block, class second_block>
class parallel_sum_pair
{
 public:
 output_t operator()(input_t input)
 {
   return _first(input) + _second(input);
 }
private:
  first_block _first;
  second_block _second;
};

template<class first_block, class second_block>
class sequence_pair
{
public:
 output_t operator()(input_t input)
 {
    return _second(_first(input));
  }
private:
  first_block _first;
  second_block _second;
};
```

Listing 5. Parallel and sequential block connectors

Larger structures could become unreadable when using nested pair connectors. Solution based on statically defined, heterogeneous container is more readable and easier to use alternative.

In the case of fixed number of blocks, instead of nested pair connectors, it is simpler to use variadic templates provided by modern revision of C++ or D language standard. *Typelists* [3] can be used as an alternative for compilers that lack support for this language feature.

If there is a need to scale number of connected elements, or the type of each block needs to be chosen algorithmically, type policies are a suitable solution. Type of each block may be determined by an index in the sequence. Such template class, passed as a template template parameter to heterogeneous container, guide compiler in memory offsetting and choosing methods that need to be applied. Logic, that determines the type for the given index needs to be defined in meta-code and evaluated during compile time. Here, template meta-programming acts as a solution. If supported, C++11 *constexpr* language feature may be used to simplify syntax. An example of type policy is provided in code listing 6.

```
template<size_t index>
class dsp_block_policy
{
public:
  typedef sequential_pair<
```

```
    blocks::delay_line<100 * index /*delay in samples*/>,
    blocks::iir_filter<2 /*poles*/, 1/*zeros*/>> type
}
```

Listing 6. Example of type policy. With use of static conditional statements and meta-programming, highly complex type choosing logic can be implemented.

As usage of such policy may not be clear, we propose a generic (not limited to DSP applications) implementation of heterogeneous container (code listing 7) that supports algorithmically defined block types. Container based on variadic template may be implemented in similar way. Having implementation of such container on hand, any type of connector can be created in a straightforward manner.

```
template<
  size_t size,
  template <size_t> class types_policy
> class meta_container
{
  // make root element accessors a friends
  template<size_t index> friend class get;
  template<
    template <size_t> class types_policy,
    size_t size,
    typename functor
  > friend void for_each(meta_container<size,
        types_policy>& container, functor& functor);

public:
  MetaContainer()
  {
    static_assert(
        size != 0,
        "meta_container␣of␣size␣0␣is␣not␣allowed."
    );
  }

private:
  meta_node<types_policy, 0, size> _root;
};

template<
  template <size_t> class types_policy,
  size_t idx,
  size_t size
> class meta_node
{
```

```
public:
  typedef std::integral_constant<size_t, idx> index;

  // type of element of this node
  typedef typename types_policy<idx>::type head;
  // tail node type
  typedef typename std::conditional<
    (size - 1 > idx),
    meta_node<types_policy, idx + 1, size>,
    null_type
  >::type tail;

public:
  head value;
  tail next;
};
```

Listing 7. Heterogenous container implementation, with type definitions provided by type policy. Instance of such container is contiguous in memory, despite linked-list like implementation, as iteration over elements takes place during compile time.

```
  template<size_t index>
  class get_node
  {
  public:
    template<
      template <size_t> class types_policy,
      size_t index,
      size_t size
    > static inline
    typename types_policy<index + typeIndex>::type&
    from(meta_node<types_policy, typeIndex, size>& container)
    {
      return get_node<index - 1>::from(container.next);
    }
  };

  // terminates template recursion
  template<>
  class get_node<0>
  {
  public:
    template<
      template <size_t> class types_policy,
      size_t index,
      size_t size
    > static inline
    typename types_policy<index>::type&
    from(meta_node<types_policy, index, size>& container)
    {
      return container.value;
    }
  };
```

Listing 8. Accessor method of meta-programming based container. Friend *get* class of meta_container delegates iteration to *get_node* classes shown in this listing, starting at the root element. Iteration is analogous to iterating over linked-lists, yet is performed during compile time. Foreach iteration has been implemented similarly to loop-unrolling code presented in listing 2 in Section 2, therefore we omit its implementation.

In the provided implementation, we can see that heterogeneous containers, such as tuples, are metaprogramming equivalent of linked-lists. Because iterating over elements is performed during compile time, there is no overhead of pointer dereferencing and cache issues typical to regular linked-list implementation. Also, type of each element is inferred by accessor methods, and many errors that would normally raise an exception in object oriented implementations occur as compilation errors, not runtime errors.

3.2.2 Composing Signal Flow via Template-Based Variant of Decorator Pattern

Template metaprogramming is enabled by *duck-typing* (naming based) paradigm enabled by template system [29], which still keeps the benefit of a fully type-safe language (types are still validated during compile time, in contrast to typical duck-typing languages like Python and Ruby). This behaviour is considered to be a case of static polymorphism [4]. Therefore, many design patterns can be mapped directly into their static equivalents [3].

In the object oriented approach, the usage of decorator pattern is considered to be an out of the box solution for pipelining computations. Templates allow to achieve the same effect without the need of using dynamic polymorphism. Therefore, in cases of sequential signal flow, simpler solution, that is based on metaprogramming implementation of decorator pattern, may be applied.

Single processing block may be defined as shown in listing 9. Such blocks compositions mimic the semantics of decorator pattern and allow to build processing pipelines.

```
template<typename prev_block>
class seq_dsp_block
{
public:
  signal_type operator()(signal_type input)
  {
    signal_type out = _prev(input);
    // signal processing code
    return out;
  }
private:
  prev_block _prev;
};
```

Listing 9. Static variant of decorator pattern applied in DSP processing block

3.3 Example of Schroeder Reverberator Implementation

As an example, we show how proposed techniques are useful in implementation of one of the first and simplest artificial reverberation algorithms, the Schroeder reverberator. Its structure is shown in Figure 2.

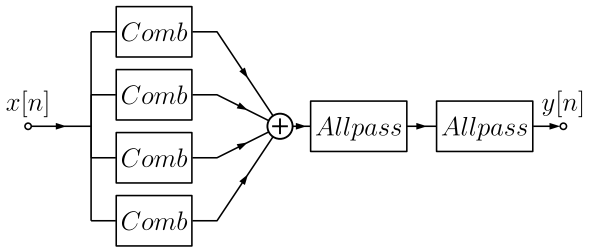


Figure 2. Schroeder reverberator block diagram

Having implementations of comb and allpass filters provided, Schroeder reverberator may be expressed as appears in code listing 10. In the provided example, blocks act as signal processing policies, defining specific behavior of generic structure (in this case, parallel and sequential groups). Since unified interface is defined as input/output type pairs, instantiated structure may act as building block for further composition, providing high level of modularity.

```
  typedef sequence<
      parallel_sum<
        comb<delay<unsigned int>>,
        comb<delay<unsigned int>>,
        comb<delay<unsigned int>>,
        comb<delay<unsigned int>>
      >,
      all_pass<delay<unsigned int>>,
      all_pass<delay<unsigned int>>
  > schroeder_rev;
```

Listing 10. Schroeder reverberator implementation

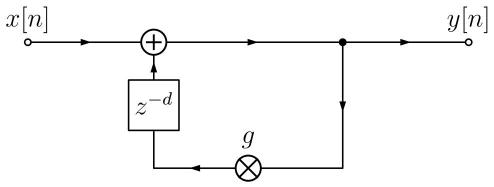


Figure 3. Comb filter block diagram

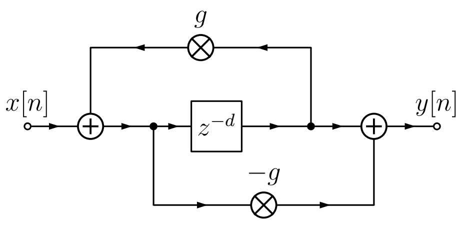


Figure 4. All-pass filter block diagram

Building blocks of Schroeder reverberator, namely comb (Figure 3) and all-pass (Figure 4) filters are simple forms of recursive structures, that due to their specifics (substantial delay of few thousand samples in typical applications) could benefit from custom implementation. Idea of block diagram decomposition may be developed further - as an example we present code listing 11, showing comb filter implementation.

```
typedef recursive<
    transparent, // forward
    sequence<scale, delay<unsigned int>> // recursive
  > comb;
```

Listing 11. Comb filter implementation

Even though such level of decomposition is possible with no impact of performance, it is generally better to provide predefined block implementations at this level of granularity.

3.4 Implementing Singe Instruction, Multiple Data (SIMD) Optimised DSP Components

SIMD instruction set usage is necessary to maximise performance of all current CPU architectures. Code vectorisation is not always a trivial task, mostly because of data dependencies and the requirement of properly aligned memory. Even though all current compilers offer auto-vectorisation feature, non-trivial cases must still be vectorised by a skilled programmer. Fortunately, as mentioned in the beginning of this section, approach described in this paper allows for heterogenous structure composition. The same principle, that allows for mixing floating/fixed point arithmetic, enables consistent SIMD processing.

Implementing DSP blocks with use of Streaming SIMD Extensions (SSE), Advanced Vector Extensions (AVX) or Advanced SIMD (NEON) instruction sets have been a subject of many studies [17, 14, 18, 23, 25]. Even vectorisation of, recursive in nature, infinite impulse response (IIR) filtration has been tacked by some [23]. For example, ITU G729C codec post processing formula transforms IIR difference equation for the samples vector (consisting of 4 or 8 consecutive samples) into the form of matrix multiplication. Techniques of DSP code vectorisation are not in scope of this paper. We focus mainly on SIMD vectorisation in context of proposed metaprogramming based implementations.

In order to create vectorised structures, each DSP block needs to operate on SIMD vector types, such as __m128 or __m256 keeping concise and elegant interface via propagation of the given type throughout series of connected blocks. The only limitation that constrains usage of SIMD instructions, results from recursive connections with delay lines shorter than number of samples processed on operational type used (e.g. 3 samples of delay when using SSE with single precision arithmetic). To avoid runtime errors, delay introduced by delay line should be fixed with statically defined value (via template parameter). Moreover, static_assert keyword should be used in order to detect errors during compilation (e.g. listing 12).

```
template <size_t delay>
class delay_line_SSE
{
  delay_line_SSE()
  {
    static_assert(
        delay >= 4,
        "delay_line_SSE: delay parameter needs to be >=4."
    );
  }

  inline __m128 operator()(__m128)
  {
    // delay line logic
  }
};
```

Listing 12. Using static asserts to detect errors during compilation

4 COMPLEX USE CASE: FEEDBACK DELAY NETWORK IMPLEMENTATION

Feedback Delay Network (FDN), being one of the most naturally sounding artificial reverberators [13, 24], became widely implemented in many sound processing software products. Although FDN is a very potent tool, achieving proper perceptual quality of acoustic simulation demands additional modifications to its structure. This applies to each of processing lines, as well as to the input/output filters. FDN needs to be fine-tuned to achieve proper acoustic properties, but having quite complex structure it demands proper level of implementation configurability. Implementation described in this section presents benefits of proposed technique, providing high level of code composability while retaining high efficiency via elimination of unnecessary polymorphic calls, heap memory allocations, SIMD vectorisation and other low level optimisations.

4.1 Brief Description FDN Reverberator Structure

The structure of FDN consists of N delay lines $DL_m = z^{(-m)}$, each resulting in a signal being delayed by $t_i = \frac{m_i}{f_s}$ seconds, where f_s is the sampling frequency. Output of these lines acts as an input to the orthogonal diffusion matrix producing output signals as well as the feedback mixed into input signal.

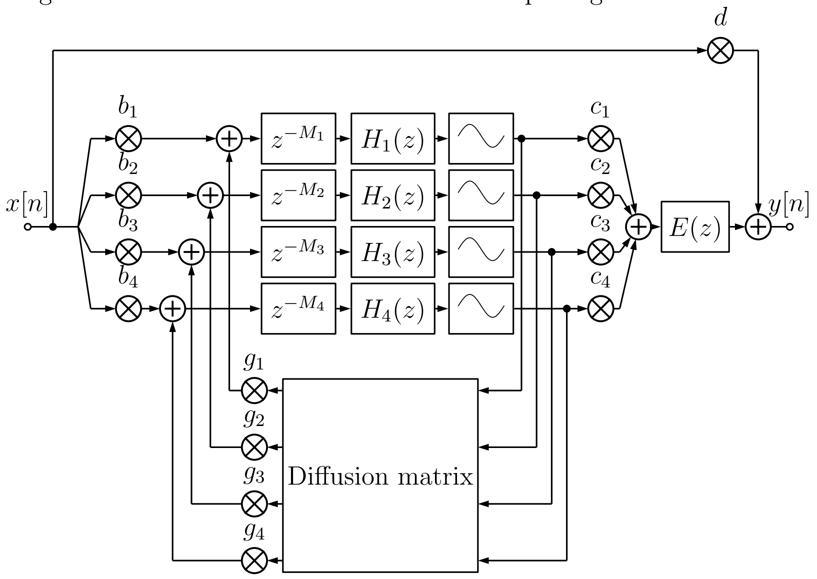


Figure 5. Feedback delay network block diagram

In the FDN block diagram (pictured in Figure 5), we can see various blocks that shall be abstracted, namely blocks within processing lines and diffusion matrix.

More advanced implementations introduce elements such as IIR filters and tone correction filters in addition to delay lines.

FDN aims to approximate acoustical environments, which may be modelled as a complex audio system with thousands of poles and zeros. Real time simulation of such system would be unacceptable in terms of computational complexity. Even though FDN is less computationally expensive, higher order networks still consume considerable amount of CPU time [13].

4.2 Achieving Modularity of FDN Implementation via Metaprogramming and Policy-Based Design

Primary motivation to use policy-based design in FDN implementation was the need to efficiently experiment with different configurations of DSP blocks within each of the processing lines, while being still able to match, and therefore assess, performance of highly optimised system. Application of object oriented design would have severe runtime impact, mostly due to virtual method calls. All the primitives described in previous section allow us to encode DSP diagram in form of nested templates. Proposed implementation is described in code listing 13.

```
template <size_t index>
class line_policy
{
public:
  typedef sequential<
    nth_prime<unsigned int /* start after */,
      index * 10 /* take each prime in strid of 10 */>::value,
    iir_filter <3 /*poles*/, 2/*zeros*/>,
    modulator<unsigned int>> type;
}

typedef parallel_sum<
  scale, // d scaling factor
  sequence<
    sum<
      parallel_vector<
        scale_vector, // b scaling factors
        recursive_vector<
          vector_of<
            line_policy,
            4
          >, // forward
          sequence_vector<
            fwht<4>, // diffusion matrix
            scale_vector, // g scaling factors
          > // backward
```

```
            >,
            scale_vector , // c scaling factors
         >,
         4
      >,
      iir_filter <3/*poles*/,2/*zeros*/>
   >> fdn_rev;
```

Listing 13. Encoded FDN diagram, according to previously discussed techniques. Notice vector processing blocks, that allow for recursive mixing of signals by the diffusion matrix (here implemented in form of fast Walsh-Hadamard transform).

5 RELATED WORK AND FURTHER DEVELOPMENT

Common DSP optimisations have been described by authors of widely used libraries. Frigo et al. [8] provided optimisation guidelines for FFT, also applicable to other *divide and conquer* derived methods. Authors focused on widely available CPU features, including OOO, super-scalar capabilities and SIMD instruction sets. SPIRAL software, described by Puschel at al. [22] addresses transforms code generation, including FFT, DWT and other.

Our approach, being more generic, is applicable to broader range of DSP algorithms. Although some authors consider the idea of DSP diagrams based automatic code generation [21, 34], we have not managed to find any recent publications that stay up to date with advancements in processor architectures.

Proposed technique efficiently solves the problem of providing fast, yet configurable DSP algorithm implementations. When used with simplified, easy to understand API, which hides the complexities of template metaprogramming, proposed techniques create a powerful framework for signal processing applications. Regardless of an exact use case, programmers can easily take an advantage of all the optimisations provided by current, highly advanced compilers.

The proposed solution may be applied in:

- Automatic translation of DSP diagrams into C++ code that is ready to compile.
- Functional paradigm domain specific languages, that are translated directly to C++.
- Visual tools, that allow for easy implementation and validation of DSP designs, allowing for measuring performance that is close to the upper limit available on the given hardware.

6 CONCLUSIONS

The solutions proposed in this paper, in spite of its unconventional characteristics, were proven to be practical in software that needs to deal with fine grained, computationally intense problems. High performance, code readability and configurability were achieved with statically defined modular architecture. Disassembly of generated software confirms that usage of metaprogramming methods leads to highly optimised machine code, that indeed looks like it has been hand tweaked by experienced assembly programmer. Moreover, being able to easily switch between SIMD implementations makes it easy to test performance on different architectures, keeping code organised and readable.

Acknowledgments

This work was supported by MNiSW resources for science as statutory activity.

REFERENCES

[1] Abrahams, D.—Gurtovoy, A.: C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond. Pearson Education, 2004.

[2] Ahmed, N.—Rao, K. R.: Walsh-Hadamard Transform. Orthogonal Transforms for Digital Signal Processing. Chapter 6. Springer, 1975, pp. 99–152, doi: 10.1007/978-3-642-45450-9_6.

[3] Alexandrescu, A.: Modern C++ Design: Generic Programming and Design Patterns Applied. Addison-Wesley, 2001.

[4] Burrus, N.—Duret-Lutz, A.—Géraud, T.—Lesage, D.—Poss, R.: A Static C++ Object-Oriented Programming (SCOOP) Paradigm Mixing Benefits of Traditional OOP and Generic Programming. Proceedings of the Workshop on Multiple Paradigm with OO Languages (MPOOL '03), Anaheim, CA, USA, 2003.

[5] Chen, J.—Joo, B.—Watson, W.—Edwards, R.: Automatic Offloading C++ Expression Templates to CUDA Enabled GPUs. 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops and Ph.D. Forum (IPDPSW), 2012, pp. 2359–2368, doi: 10.1109/IPDPSW.2012.293.

[6] Driesen, K.—Hölzle, U.: The Direct Cost of Virtual Function Calls in C++. ACM Sigplan Notices, Vol. 31, 1996, No. 10, pp. 306–323, doi: 10.1145/236337.236369.

[7] Fog, A.: Optimizing Software in C++: An Optimization Guide for Windows, Linux and Mac Platforms, 2004.

[8] Frigo, M.—Johnson, S. G.: FFTW: An Adaptive Software Architecture for the FFT. Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, 1998, Vol. 3, pp. 1381–1384, doi: 10.1109/ICASSP.1998.681704.

[9] Frigo, M.—Johnson, S. G.: The Design and Implementation of FFTW3. Proceedings of the IEEE, Vol. 93, 2005, No. 2, pp. 216–231, doi: 10.1109/JPROC.2004.840301.

[10] Gamma, E.—Helm, R.—Johnson, R.—Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education, 1994.

[11] Gawlik, I.—Pędzimąż, T.—Pałka, S.—Ziółko, B.: Efficient Vectorized Architecture for Feedback Delay Network Reverberator with Policy Based Design. Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA), Poznan, 2015, pp. 124–127.

[12] Hammarlund, P.—Martinez, A. J.—Bajwa, A. A.—Hill, D. L.—Hallnor, E. et al.: Haswell: The Fourth-Generation Intel Core Processor. IEEE Micro, Vol. 34, 2014, No. 2, pp. 6–20, doi: 10.1109/MM.2014.10.

[13] Jot, J.-M.—Chaigne, A.: Digital Delay Networks for Designing Artificial Reverberators. Audio Engineering Society Convention, AES, Vol. 90, 1991, Art. No. 3030.

[14] Lee, J.—Moon, S.—Sung, W.: H.264 Decoder Optimization Exploiting SIMD Instructions. Proceedings of the 2004 IEEE Asia-Pacific Conference on Circuits and Systems (APCCAS), 2004, Vol. 2, pp. 1149–1152.

[15] Mahapatra, N. R.—Venkatrao, B.: The Processor-Memory Bottleneck: Problems and Solutions. Crossroads – Computer Architecture, Vol. 5, 1999, No. 3es, Art. No. 2.

[16] McCartney, J.: Rethinking the Computer Music Language: SuperCollider. Computer Music Journal, Vol. 26, 2002, No. 4, pp. 61–68, doi: 10.1162/014892602320991383.

[17] Nguyen, H.—John, L. K.: Exploiting SIMD Parallelism in DSP and Multimedia Algorithms Using the AltiVec Technology. Proceedings of the 13th International Conference on Supercomputing (ICS '99), ACM, 1999, pp. 11–20, doi: 10.1145/305138.305150.

[18] Norilo, V.—Laurson, M.: Kronos – A Vectorizing Compiler for Music DSP. Proceedings of the 12th International Conference on Digital Audio Effects (DAFx-09), 2009.

[19] Orlarey, Y.—Fober, D.—Letz, S.: FAUST: An Efficient Functional Approach to DSP Programming. In: Assayag, G., Gerzso, A. (Eds.): New Computational Paradigms for Computer Music. IRCAM, 2009.

[20] Plauger, P. J.—Stepanov, A. A.—Lee, M.—Musser, D.: C++ Standard Template Library. Prentice Hall PTR, 2000.

[21] Powell, D. B.—Lee, E. A.—Newman, W. C.: Direct Synthesis of Optimized DSP Assembly Code from Signal Flow Block Diagrams. 1992 IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-92), Vol. 5, 1992, pp. 553–556, doi: 10.1109/ICASSP.1992.226560.

[22] Puschel, M.—Moura, J. M.—Johnson, J. R.—Padua, D.—Veloso, M. M.—Singer, B. W.—Xiong, J.—Franchetti, F.—Gacic, A.—Voronenko, Y.—Chen, K.—Johnson, R. W.—Rizzolo, N.: SPIRAL: Code Generation for DSP Transforms. Proceedings of the IEEE, Vol. 93, 2005, No. 2, pp. 232–275, doi: 10.1109/JPROC.2004.840306.

[23] Robelly, J. P.—Cichon, G.—Seidel, H.—Fettweis, G.: Implementation of Recursive Digital Filters into Vector SIMD DSP Architectures. IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP '04), 2004, Vol. 5, pp. 165–168, doi: 10.1109/ICASSP.2004.1327073.

[24] Rocchesso, D.—Smith, J. O.: Circulant and Elliptic Feedback Delay Networks for Artificial Reverberation. IEEE Transactions on Speech and Audio Processing, Vol. 5, 1997, No. 1, pp. 51–63, doi: 10.1109/89.554269.

[25] Shahbahrami, A.—Juurlink, B.—Vassiliadis, S.: Efficient Vectorization of the FIR Filter. Proceedings of the 16th Annual Workshop on Circuits, Systems and Signal Processing (ProRISC), 2005, pp. 432–437.

[26] Sipos, Á.—Porkoláb, Z.—Pataki, N.—Zsók, V.: Meta⟨Fun⟩ – Towards a Functional-Style Interface for C++ Template Metaprograms. Proceedings of 19th International Symposium of Implementation and Application of Functional Languages (IFL 2007), 2007, pp. 489–502.

[27] Stewart, T.: Intel Integrated Performance Primitives: How to Optimize Software Applications Using Intel IPP. Intel Press, 2004.

[28] Stroustrup, B.: The Design and Evolution of C++. Pearson Education India, 1994.

[29] Stroustrup, B.: Foundations of C++. In: Seidl, H. (Ed.): Programming Languages and Systems (ESOP 2012). Springer, Lecture Notes in Computer Science, Vol. 7211, 2012, pp. 1–25, doi: 10.1007/978-3-642-28869-2_1.

[30] Veldhuizen, T.: Expression Templates. C++ Report, Vol. 7, 1995, No. 5, pp. 26–31.

[31] Veldhuizen, T. L.: C++ Templates Are Turing Complete. Available at `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.14.3670&rep=rep1&type=pdf`, 2003.

[32] Wang, G.: The Chuck Audio Programming Language. A Strongly-Timed and On-the-Fly Environ/Mentality. Ph.D. dissertation, Princeton University, 2008.

[33] Wang, G.—Cook, P. R.: ChucK: A Concurrent, On-the-Fly, Audio Programming Language. Proceedings of the 2003 International Computer Music Conference, 2003, pp. 219–226.

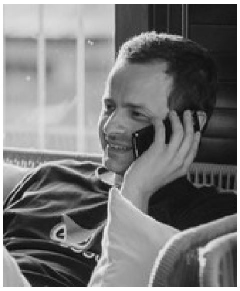
[34] Wess, B.—Kreuzer, W.: Optimized DSP Assembly Code Generation Starting from Homogeneous Atomic Data Flow Graphs. Proceedings of the 38th Midwest Symposium on Circuits and Systems, 1995, Vol. 2, IEEE, pp. 1268–1271, doi: 10.1109/MWSCAS.1995.510327.

[35] Ziółko, B.—Pędzimąż, T.—Pałka, S.—Gawlik, I.—Miga, B.—Bugiel, P.: Real-Time 3D Audio Simulation in Video Games with RAYAV. Making Games, Vol. 1, 2015.

Ireneusz Gawlik received his M.Sc. degree in acoustical engineering and B.Sc. in computer science from the AGH University of Science and Technology in Kraków, Poland. Currently he is working toward his Ph.D. degree in computer science at the AGH UST. His research is focused on creating models of acoustic phenomena and development of audio processing algorithms. He is also working on development of machine learning algorithms, machine learning in BigData setups, recommender systems and learning to rank.

Szymon Pałka received his M.Sc. degree in computer science from the AGH University of Science and Technology in Kraków, Poland. Currently he is working toward his Ph.D. degree in computer science at the AGH UST. His research is focused on optimization of geometric techniques for 3D scene analysis for spatial audio processing. He is a co-author of a patent application and scientific papers regarding simulation of sound propagation and speech processing. He teaches "Computer Graphics" class at the AGH University.

Tomasz Pędzimąż received his M.Sc. degree in computer science from the AGH University of Science and Technology in Kraków, Poland. Currently he is working toward his Ph.D. degree in computer science at the AGH UST. He is an author or co-author of 10 scientific papers and 2 patent applications, with one patent granted. He has participated in several national and European research projects. His research is focused on natural language processing regarding speech recognitions language models. He teaches "Computer Graphics" class at the AGH University.



Bartosz Ziółko studied electronics and telecommunications at the AGH University of Science and Technology in Krakow. Next he did his Ph.D. in computer science at the University of York. He is an author or co-author of over 100 scientific papers and of 3 patent applications, with two patents granted. He is the main author of book "Przetwarzanie mowy" (Eng. Speech Processing). His research interests include automatic speech recognition, natural language modeling, speaker recognition and soundtracing. He is CEO and a co-founder of Techmo – an AGH spin-off company and Assistant Professor at AGH University of Science and Technology. He has participated in several national and European research projects. He was also engaged as external consultant in speech technologies for companies. His R & D activity resulted in a few products licensed to companies, universities and Court. He is also governmental technology expert. He was trained in research commercialization by Stanford and in Science Infrastructure Management by IBM and Fraunhofer Institute.