Computer Science and Information Systems 11(4):1499–1514 DOI: 10.2298/CSIS130224080D

Language Abstractions For Low Level Optimization Techniques

Gergely Dévai, Zoltán Gera, and Zoltán Kelemen

Faculty of Informatics, Eötvös Loránd University,
Pázmány P. stny. 1/C, 1117 Budapest, Hungary
{deva,gerazo,kelemzol}@caesar.elte.hu

Abstract. Programmers are often forced to implement performance-critical applications at a low abstraction level. This leads to programs that are hard to develop and maintain because the program text is mixed with low level optimization tricks and is far from the algorithm it implements.
Even if compilers are smart nowadays and provide the user with many automatically applied optimizations, practice shows that in some cases it is hopeless to optimize the program automatically without the programmer's knowledge. A complementary approach is to allow the programmer to fine tune the program by providing him with language features that make the optimization easier. These are language abstractions that make optimization techniques explicit without adding too much syntactic noise to the program text.
This paper presents such language abstractions for two well-known optimizations: *bitvectors* and *SIMD* (Single Instruction Multiple Data). The language features are implemented in the embedded domain specific language Feldspar which is specifically tailored for digital signal processing applications. While we present these language elements as part of Feldspar, the ideas behind them are general enough to be applied in other language definition projects as well.

Keywords: Optimization, bitvector, SIMD.

1. Introduction

The abstraction level of programming languages has dramatically increased during the journey from assembly to modern functional and modeling languages. On the other hand, more abstraction usually means less programmer control over platform specific features of the code. In theory, optimizing compilers can solve this problem and generate object code that optimally uses the memory structure and the instruction set of a given platform. However, practice shows that there are many application areas where low abstraction level languages (usually C or even assembly) are used either to support special hardware or to meet the strict runtime performance and memory consumption requirements. This is one reason for the fact that C is usually in the first 3 most popular languages (not rarely on the first place) and even assembly is among the most popular 10-20 according to statistics like the Tiobe Index [8] or LangPop [7].

Low abstraction level is a risk from the maintenance point of view. Code is more verbose, and it reflects more how data is stored and computation is performed instead of what is stored and computed. Therefore it is harder to understand and to change. This situation is even worse if the source code is optimized for a given hardware-software

ecosystem: Uses special hardware instructions to make computation faster, changes data layout according to cache lines, rearranges instructions to hide memory access overhead, changes according to the optimization capabilities of a given compiler.

These are examples of what we will refer to as *low level optimization* in this paper. The main objective of this paper is to find ways to give control to the programmer over these optimizations while preserving the maintainability of the software. Our our approach is to create domain specific language features, suitable *abstractions* for given optimization techniques in order to combine easy-to-maintain source code with efficient target code and to simplify the supporting compiler modules. We will do so by exploring two specific optimization techniques: bitvectors and SIMD, and presenting corresponding libraries of the Feldspar domain specific programming language. Using these libraries induces minimal changes compared to the unoptimized source code while increases the runtime performance considerably. Finally, the paper identifies the key ideas behind these libraries to generalize the results and make them applicable in future language implementation projects.

The paper is organized as follows. The next section presents the two optimization techniques to be used throughout the paper as example, and also gives a brief introduction to our implementation language. Section 3 presents language abstractions for bitvectors. It first explains the concept using a simple algorithm then proceeds with a real-world example: cyclic redundancy check, finally discusses the implementation of the library. Section 4 uses the scalar product algorithm as the example to present the abstractions for SIMD optimization, then introduces the most important aspects of the corresponding library implementation. Section 5 compares our method to related work, while the final section concludes the paper by pointing out the most important aspects of our solution.

2. Technical background

This section introduces the concept of two widely used optimization techniques, bitvectors and SIMD, that will be used as examples later on in the paper. The Feldspar language, used for the implementation of our concepts, is also presented.

2.1. Bitvectors

The *smallest addressable data types* of all platforms are larger than a single bit, so general programming languages usually use a whole *word* (which is the most conveniently accessible data type of the platform) for storing a bool. This is not a problem when boolean values are used only in condition evaluations, but implementing algorithms that process series of boolean values is usually challenging:

– Using the *vector* construction of a given language with boolean values usually leads to the use a whole word of the hosting platform per boolean. This means 8-times to 64-times more *memory consumption* compared to what would be required. Even if accessing a word seems to be faster then accessing a single bit, the increased memory footprint and decreased locality can impact cache performance so seriously that it will become the main performance bottleneck causing overall speed degradation [15].

– Writing *optimized code* involves *packing* bits into other larger words, getting and setting bits by *shifting* and *masking*. The resulting code no longer shows the original abstractness of the algorithm, which is harder to understand and maintain, and engineers capable of touching this code cost far more [20].

Usually one of the above mentioned approaches is favored over the other depending on the exact real world context without recognizing the need to find a better, more general solution. In order to get one step closer to this general solution, one can use libraries which implement *bitvectors*. One such standard library is the *Standard Template Library* of *C++* (STL) which contains a specialized implementation of the vector container dedicated to bools: `vector<bool>` [19]. Its implementation solves the storage issue while keeping the code in an easily maintainable shape. However, boolean operations can be done in one step on all bits placed in the same word leading to 8-times to 32-times performance boost. Unfortunately STL's `vector<bool>` cannot guarantee such optimizations.

In order to get this performance boost, the underlying bitvector implementation should be clever enough to find out when operations can be transformed into one single compound operation executed on multiple bits at once. One way is to involve difficult code analysis techniques and have some of the typical code patterns transformed into more efficient code. Another way is to let the programmer use special constructions to instruct the compiler where to apply such optimizations. This way the programmer is in control instead of relying on unpredictable compiler optimizations. This approach is used by *Feldspar*.

2.2. Single Instruction, Multiple Data

The technique to increase performance by doing instructions at once on multiple data is called *SIMD* (Single Instruction, Multiple Data). We organize more than one piece of data into packets and try to do operations on the packet itself. To be precise, there are two ways of processing multiple data in parallel on one CPU core:

– The *SIMD* way is to use large registers to store multiple smaller values and support special instructions which operate on such registers capable of doing parallel primitive operations on the individual values. The drawback is that every operation paired with every possible operand size needs a separate instruction to be implemented in hardware resulting in an instruction boom giving more complicated hardware [18]. It is typical to have from 32-bit up to 128-bit SIMD capable registers giving the chance to *pack* 2, 4, 8 or 16 values in one register depending on the type used.
– The *MIMD* way (Multiple Instruction, Multiple Data) is used when one has the chance to execute multiple instructions parallel on the same core. This means explicitly issuing parallel instructions coming from one thread of execution and should not be confused with hyper-threading techniques. Standard *CISC* (Complex Instruction Set Computer) and *RISC* (Reduced Instruction Set Computer) processors do not support this style of execution. Only *VLIW* (Very Long Instruction Word) processors have this capability like the *TMS320C6000* family of *DSP processors* which have 8 parallel *execution units* per core. The advantage of this technique is that it does not need special instructions to meet our needs. The disadvantage is that it is hard to compile programs which can continuously use all parallel units of the CPU [10].

The *SIMD* way is to compile vector operations into platform dependent code which is built of a specific SIMD instruction set. The *x86/x64* platform has *SSE*, *ARM* has *NEON*, *MIPS* has *DSP ASE*, and *PowerPC* has *AltiVec* as its SIMD instruction set.

The different methods of compiling SIMD capable code are:

- SIMD *intrinsics* of the target platform can be generated into the *C* source to have SIMD instructions compiled. All platforms have completely different intrinsics, so this increases the compiler's complexity but gives the full potential of the platform's SIMD capabilities.
- *GCC*, from version *4.2*, supports automatic vectorization for some platforms and has some special attributes to generate SIMD capable vector types by hand [17]. This support generally fits the *C* language, so this does not give a solution for operations which do not have their non-SIMD *C* language counterpart. Examples for such operations are dot-product or summing of values inside a vector. Functionality like these should still be generated by hand (using intrinsics).
- *LLVM* does support vector types which are the best candidates for an LLVM backend to compile platform specific SIMD instructions from LLVM bytecode [21]. However, this has the same problems as the previous approach.

Current approaches [22] [16] [13] for SIMD code generation concentrate only on a specific problem or algorithm. In contrast, our method presented in section 4 benefits from the extra information given in the program code of a domain specific language. By using these language abstractions we can give a more general solution with less difficult heuristics compared to the referenced approaches.

2.3. Feldspar

Feldspar [9] is a domain specific language for digital signal processing (DSP). It is an embedded language in Haskell: it consists of a set of special Haskell libraries. The language is a functional one. It reimplements a selection of Haskell idioms that are useful for DSP applications and can be compiled efficiently to the target object languages, currently C99 and LLVM.

Feldspar has a *core language* consisting of arithmetic, logic and bit operations, conditional branching and loop-like constructs to process sequences of data. The compiler of the language translates this core to the target languages. Most of the language features are implemented as lightweight libraries built on top of the core language. This modular design allows one to create new language features without modifying the core compiler. Both language features presented in this paper are of this kind.

The most important language abstraction on top of the Feldspar core language is the *Vector* library. Many standard list processing functions of Haskell are redefined by Feldspar to work on vectors. For example, the `tail` function removes the first element of a vector. Another example is `zipWith` that takes a function of arity two and two vectors. It *zips* the two vectors, i.e. it forms pairs of the elements located at the same index and applies the function on these pairs. If the arguments of `zipWith` differ in length, the length of the result will be that of the shorter input, and the extra elements at the end of the longer input are discarded. The *Vector* library guarantees an optimization called *vector fusion*: an arbitrarily complex vector expression is *fused* into a single loop. For

example, the expression `zipWith (+) v (tail v)` produces a single loop with the instruction `result[i] = v[i] + v[i+1]` inside.

By design, the elements of Feldspar's vectors can be computed independently. This allows back-ends to parallelize these computations. In addition to this, Feldspar's latest version has support for task parallelism. Note, however, that none of the optimizations discussed in this paper require thread-parallel execution: these are techniques to be applied on sequential portions of the code.

3. Language Abstraction for Bitvectors

We have created a *BitVector* library for Feldspar that provides the user with types like `BitVector Word32`. The programmer can manipulate a bitvector of this type similarly to a sequence of boolean values, while the compiled code stores 32 booleans in each 32-bit word and processes the bitvector word-by-word instead of bit-by-bit.

3.1. Introductory example

For example let us write a function that takes a sequence of booleans and produces a sequence of booleans such that the i^{th} element of the result is $true$ if and only if both values at positions i and $i + 1$ in the input are $true$. Figure 1 shows the implementation using the original $Vector$ library of Feldspar.

```
vec1 :: Vector (Data Bool) -> Vector (Data Bool)
vec1 v = zipWith (&&) v (tail v) ++ vector [False]
```

Fig. 1. First implementation using the *Vector* library

Functions `tail` and `zipWith` work as discussed before in section 2.3. The function `(++)` is used to concatenate two vectors. In this case, the single element vector `vector [False]` is appended the end to keep the original size.

Let us compile this function to C99 and run it with an input of 256 boolean values. The architecture we used stores each boolean value in one byte, therefore the input and output arrays need half kilobyte of memory in total. The resulting loop will perform 255 iterations with the instruction
`result[i] = v[i] && v[i+1]`
and an extra iteration with
`result[255] = v[255] && false`.
This solution wastes memory and runs slowly.

Using the vector library one can boost the performance only by lowering the abstraction level considerably. This solution is shown on figure 2.

This solution uses vectors of 32-bit words and the logic of the original algorithm on boolean sequences is enforced using bitwise conjunction (`.&.`), disjunction (`.|.`) and shifting (`shiftLU`, `shiftRU`) operations. The performance of this solution is much better than the previous one: in order to process 256 boolean values, the input and output

```
vec2 :: Vector (Data Word32) -> Vector (Data Word32)
vec2 v = zipWith (.&.) v v' ++ end
  where
    v' = zipWith (.|.)
             (map (`shiftLU` 1) v)
             (map (`shiftRU` 31) (tail v))
    end = indexed 1
             (\i -> last v .&. (last v `shiftLU` 1))
```

Fig. 2. Optimized implementation using the *Vector* library

arrays consume only 64 bytes of memory in total. The loop in the resulting C code performs
`result[i] = v[i] & ((v[i] << 1) | (v[i+1] >> 31))`
seven times and sets the last word as
`result[7] = v[7] & (v[7] << 1).`
On the other hand this solution is not easier to understand and maintain than the corresponding C code.

The *BitVector* library is able to compose the readability of the first solution with the performance of the second one by making the solution in Figure 3 possible.

```
bitvec :: BitVector Word32 -> BitVector Word32
bitvec v = zipWith (&&) v (tail false v)
```

Fig. 3. Solution using the *BitVector* library

The code is almost the same as that of the first solution except that we use bitvectors instead of vectors and the `tail` function takes an extra boolean argument instead of explicit concatenation. The reason for the latter difference is that each operation of the $BitVector$ library has to keep the length of the boolean sequence divisible by the word length. Therefore the `tail` operation, while removing the first element, appends the extra boolean argument to the end of the sequence. The generated C code is almost the same that the one from the second, low level solution.

Table 1 shows the run times in microseconds for the three presented solutions working on 256-element arrays. The results shown are obtained by calculating the average execution time over 1000 runs for each of the algorithms.

Runtime performance measurements were carried out on an *Intel Core i5-460M CPU* running at 2.53 GHz and having 3MB of L3 cache. The machine is running *Linux* with *kernel 2.6.38-8* and *gcc 4.5.2*.

The Feldspar programs were compiled using the 0.4.0.2 version of the Feldspar compiler. The source and generated codes as well as the `main` functions used for the performance measurements can be downloaded from [5].

	vec1	vec2	bitvec
default	2.334 μs	0.123 μs	0.122 μs
-O3	0.956 μs	0.014 μs	0.015 μs

Table 1. Running times of the vector and bitvector solutions

3.2. Cyclic Redundancy Check

Cyclic redundancy check (CRC) algorithms are used to compute a fixed length checksum of arbitrary long bit sequences. The checksum is used as error-detecting code or sometimes as a hash function.

The algorithm is based on binary polynomial division: The input sequence is divided by a given control polynomial and the remainder is the result. The input is first padded with zero bits at the end, then a window is applied to the beginning of the padded input. In each iteration, the window is shifted one bit towards the end of the input. If the bit leaving the window is 1, the new window contents are *xor*-ed with the control bits, otherwise it is left unchanged. This algorithm is easy to implement in terms of boolean sequences, see Figure 4.

```
crc_bool :: Vector1 Bool -> Vector1 Bool -> Vector1 Bool
crc_bool input control
  = fold step (take n padded) $ drop n padded
  where
    n = length control
    padded = input ++ repl n false
    step window bit = head window ?
        ( zipWith (/=) (tail window ++ repl 1 bit) control
        , tail window ++ repl 1 bit
        )
    repl n val = indexed n $ const val
```

Fig. 4. CRC implementation using boolean vectors

The iteration is governed by the *fold* function that executes *step* repeatedly. The initial value for this loop (i.e. the first contents of the window) is the first n bits (`take n padded`), and each iteration processes one more bit from the remaining padded input (i.e. `drop n padded`).

Each step examines the first bit in the window (`head window`) to decide if the exclusive or operation (`zipWith (/=)`) is needed or not. The window shift is implemented by removing its first bit (`tail window`) and appending the next bit of the input at the end (`++ repl 1 bit`).

This implementation is easy to write and understand, but it is way too inefficient to be useful in practice since each bit is represented as a boolean value stored in at least one byte. In addition to the storage problem, the exclusive or operation is done bit-by-bit instead of performing a bitwise xor operation on 8, 16 or 32 bits depending on the CRC width.

We use the $BitVector$ library again to optimize this algorithm while keeping its clarity. Figure 5 shows the 16 bit version.

```
crc_bitvector
  :: BitVector Word16 -> Data Word16 -> Data Word16
crc_bitvector input control
  = fold step (first padded) $ dropUnits 1 padded
  where
    padded = input ++ replUnit 1 0
    step window bit = first $ head window' ?
        ( zipWith (/=) (tail bit window') control'
        , tail bit window'
        )
      where
        window' = single window
        control' = single control
```

Fig. 5. CRC implementation with bitvectors

Instead of the boolean vectors of the original implementation, here the input is a bitvector represented as a sequence of 16 bit words. The control polynomial and the result are 16 bit words.

The structure of the iteration, the padding and the implementation of the *step* function closely follow the original code. The reason for some of the differences is that the state of the iteration (the window) is of type `Data Word16` instead of a vector. For this reason helper functions are used to get the first word of a bitvector (`first`) and to convert a word to a bitvector (`single`). The functions `dropUnit` and `replUnit` are provided by the *BitVector* library to remove a given number of complete words from the beginning of a bitvector and to create a bitvector by replicating a word a given number of times. The special `tail` function for bitvectors, accepting two parameters, has already been discussed earlier, in section 3.1.

The performance measurements were carried out the same way as described in section 3.1, except that the Feldspar compiler version was 0.6.0.2 in this case. The presented implementations were run on 160 bit input and 16 bit control vectors. Zero and one bits were evenly distributed in both sequences to make all branches of the algorithm evenly executed. Averaging over 1000 repeated executions, the boolean version takes 53 microseconds to execute while the bitvector version takes this figure down to 22 microseconds, yielding a speedup of 241%.

The reason for the huge speedup is that the bitvector solution performs 16 times less iterations compared to the one using boolean vectors. This suggests a 16 times speedup, but we also have to consider that a single iteration of the bitvector solution is heavier than in case of boolean vectors due to the shifting and masking operations involved.

3.3. Implementation

The *BitVector* library provides a new container type in Feldspar, an alternative to the original *Vector* library. It reimplements the most important vector operations and extends this set by bitvector-specific functionality. The library has been released with Feldspar and can be downloaded from the Hackage library database [4].

The most important feature of Feldspar's vectors is loop fusion. It does not matter how many operations are composed to process the input vector, the generated C code implements these using a single loop without the need for temporary storage. There is a well defined set of exceptions: operations that actually compute the vector elements and store them in memory. This way the optimization is guaranteed to happen and the result is well predictable, in contrast to traditional compiler optimization techniques.

Feldspar's vector fusion relies on the representation of vectors, which is basically composed of its length and an index function. The index function tells how to compute the element of the vector at a given index. Each time a vector operation is used to transform the vector, the index function is modified accordingly. When the vector is finally computed and written to memory, a single loop iterates through the index set and the loop body evaluates the index function.

The *BitVector* library provides the same fusion technique and extends it with the capability of processing 8, 16 or 32 bits together, even if the algorithm was defined in a bit-by-bit manner.

```
data BitVector w
     = BitVector
     { segments :: [Segment w]
     }

data Segment w
    = Segment
    { numUnits :: Data Length
    , elements :: Data Index -> Data w
    }
```

Fig. 6. Data types for the representation of bitvectors

As seen in Figure 6, a `BitVector` is a list of segments. This is necessary to be able to implement concatenation efficiently: concatenated bitvectors will become the segments of the result. Each segment is composed of the number of words in that particular segment (`numUnits`) and an index function from indexes to words. This is the key to implement the fusion functionality discussed above.

Both data types are parametric over the word length. The `w` type parameter can be instantiated by `Word8`, `Word16` and `Word32`.

Implementation of the bitvector specific optimizations are well demonstrated by the definition of the `map` function, depicted in Figure 7.

The purpose of the `map` function is to transform each element of the input vector (`bv`) using a given function (`f`) which, in case of bitvectors, is a bool-to-bool function.

```
map :: (Data Bool -> Data Bool) -> BitVector w -> BitVector w
map f bv = boolFun1 f result
  where
    result f' =
      BitVector $
        Prelude.map (\s -> s{elements = f' . elements s})
                    (segments bv)
```

Fig. 7. Implementation of the `map` function for bitvectors

The helper function `boolFun1` is used to *lift* the function `f` to a word-to-word function. For example, logical negation is lifted to the bitwise not operation on words. The lifted function is denoted as `f'` and is composed with the index functions of the input bitvector's segments to form the segments of the result.

Index function transformation and the lifting of boolean functions are the two ideas that drive the implementation of all bitvector operations. Note however, that many of them are tricky to implement. The tail function, for example, has to remove the first bit of the sequence. In order to do this the transformed index function has to compute each word of the bitvector out of two consecutive words of the original bitvector using shifting and masking operations. The complexity of the `zipWith` operation, on the other hand, is caused by the fact that the segmentation of the zipped bitvectors may differ.

4. Language Abstraction for SIMD

We have created a Feldspar library that allows SIMD optimization at a high abstraction level. This library provides types like *SIMD4 Int16* meaning that four *Int16* values are to be processed together.

We will use scalar product as an example. Let us first implement it without any optimization, using the standard *Vector* library only. The result is shown in Figure 8.

```
scalarProd :: Vector1 Int16 -> Vector1 Int16 -> Data Int16
scalarProd a b = sum (zipWith (*) a b)
```

Fig. 8. Original scalar product implementation

We multiply the corresponding elements of two vectors and sum the results. The Feldspar compiler transforms this to a single loop that makes as many iterations as the number of elements in the vectors.

In order to make this code more efficient on a target platform that supports SIMD operations, we may pack multiple elements to be processed together. Figure 9 shows this optimized solution.

There are two changes compared to the original code. Instead of *Int16*s we use *SIMD4 Int16*s as vector elements and as a last step of the computation we have to call `sum4`

```
scalarProdSIMD
  :: Vector (SIMD4 Int16) -> Vector (SIMD4 Int16) -> Data Int16
scalarProdSIMD a b = sum4 (sum (zipWith (*) a b))
```

Fig. 9. Scalar product implementation using the SIMD library

(one of the SIMD-specific operations provided by the library) to get the final result. The latter change is needed because the multiplication in the first parameter of `zipWith` and the addition behind `sum` are overloaded such that they work on SIMD packets. For this reason the result of the summation is a packet of four integers. In order to get the final result, `sum4` is used to add these four numbers.

	`scalarProd`	`scalarProdSIMD`	speedup
default	517 μs	468 μs	10.5%
-O3	227 μs	172 μs	32%

Table 2. Running times of the scalar product implementations.

Table 2 summarizes the running times of the two presented implementations working on vectors of 100000 elements. The results shown are obtained by averaging over 1000 runs of the algorithms. The performance measurements were carried out the same way as described in section 3.1.

4.1. Implementation

There is a notable difference between the design of the two presented libraries. Bitvectors overload vector operations and are used instead of vectors. SIMD types overload standard integer arithmetic and are typically used together with the standard vector library.

```
data (Packable2 a) => SIMD2 a = SIMD2 (Data (Pack2 a))

data (Packable4 a) => SIMD4 a = SIMD4 (Data (Pack4 a))

data (Packable8 a) => SIMD8 a = SIMD8 (Data (Pack8 a))
```

Fig. 10. Definition of the *SIMD* types.

Figure 10 shows the definition of the SIMD types provided by the library. Each of them is parametric with a *packable* constraint on the type variable `a`. For example, `Int8` and `Int16` are instances of the `Packable4` type class, because the targeted architectures support packing four of them together. `Pack4` is a type function that tells the size of the packets. For example:

```
type Pack4 Int16 = Int64
```

According to this, `Vector (SIMD4 Int16)` is isomorphic to a vector of 64 bit integers. However, arithmetic on `SIMD4 Int16` elements and on `Int64` values work differently. An addition instruction on two `SIMD4 Int16` values `a` and `b` generate the `feldspar_4x16_add(a,b)` expression in the C code, where `feldspar_4x16_add` is an inline function having platform- and/or compiler-specific implementation.

```
typedef int16_t v4hi __attribute__ ((vector_size (8)));

static inline
int64_t feldspar_4x16_add( int64_t arg1, int64_t arg2 )
{
    v4hi temp = ((* (v4hi*) &arg1) + (* (v4hi*) &arg2));
    return *((int64_t*) & temp);
}
```

Fig. 11. GCC-specific implementation of a SIMD addition operation

Figure 11 shows the implementation of `feldspar_4x16_add` in case of using *gcc* specific language extensions. The type `v4hi` is defined to be 8 bytes long by the attribute `vector_size (8)` and containing `int16_t` elements. The generated Feldspar code uses 64 bit integers that are passed to the `feldspar_4x16_add` function. These parameters are type casted to `v4hi` SIMD packets to perform the arithmetic needed, then the result is casted back to `int64_t`.

Interested readers are referred to [5] in order to access the source code of the SIMD library.

5. Related Work

The most popular optimization related language features are pragmas and pragma-like keywords that control some aspects of how the compiler works. Some of these, like the *register* and *align* keywords of C [2] or the *Default_Storage_Pool* pragma of Ada [3] control where data is stored. The *inline* pragma is present in both languages to give hints to the compiler which function definitions to inline at the call sites. The *unroll* pragma in C can be used to advise the compiler how to unroll the annotated loops.

One can also find pragma-like features in logic and functional languages. The *cut operator* (*!*) of Prolog [1] is used to make programs more efficient by restricting the backtrack algorithm performed by the Prolog runtime environment. In Haskell [12], the very same symbol is the *stricktness flag* that controls which parts of data structures are to be eagerly evaluated to weak head normal form when a piece of data of a specific type is created. There are compiler-specific marks as well: Unboxed types, using the # symbol, are provided by the GHC Haskell compiler [6] to ensure that primitive data types are stored by their raw value and not through a pointer on the heap.

On the positive side, pragmas does not affect the readability and maintainability of programs much while they allow fine tuning the performance on a given platform. On the other hand, they are limited in the sense that they only switch on and off standard optimization algorithms built into the compiler for the affected fragments of the program. Using pragmas it is usually not possible to achieve transformations necessary to turn a program operating on a boolean array to one that operates on a bitvector and compress 8 to 64 iterations of the original code into a single instruction. Even if a given optimization can be activated by a pragma, in many cases it is not *guaranteed* to happen. It is up to the power of the static analyzer algorithms built into the compiler if the requested transformation actually happens. In summary, our method can be used to implement more powerful and more predictable optimizations.

There are many programming language elements that support modularization and moving complexity from business logic code to libraries: package system, classes with encapsulation, templating mechanisms etc. These can be used to hide highly optimized implementations from the users of the library. A good example, closely related to our paper, is the `vector<bool>` specialized C++ template class, which provides the compact representation of bitvectors. On the other hand, it cannot achieve loop fusion and word-based processing of bitvectors that our solution guarantees.

Another question is the cost of implementation of optimization related language features. Without any particular support from the compiler architecture, the compiler itself is to be modified to add new syntax or to affect how code generation is done. An interesting method to make language extensions less costly is *OpenC++*. This language is based on C++ and, in addition, provides meta level features to define language extensions. The implementation of an extension consists, on one hand, of a meta language program defining how to compile the language extension to regular C++, and, on the other hand, of runtime support code. The OpenC++ compiler transforms the source program according to the defined extensions to C++ code that should be linked with the runtime support code.

OpenC++ can be used to implement language extensions for optimization purposes that are more complex and powerful than pragmas and traditional libraries. For example, the OpenC++ tutorial [11] shows how to implement operators for a matrix library that can do the same kind of fusion optimization that Feldspar Vectors are designed for (see Section 2.3).

The main difference compared to our approach is that OpenC++ language extensions are defined in terms of manipulations of the abstract syntax tree, while we exploit the potential in language embedding to achieve the power of syntax tree manipulation in a much lighter weight form, that is similar to traditional library implementation.

6. Conclusion

Sections 3 and 4 have shown two examples for language abstractions that provide considerable performance improvements while not degrading the readability and maintainability of the programs. All this required only a moderate implementation effort to add support for the new language features to the compiler in a modular way, without modifying the core Feldspar compiler.

There are three ingredients of our solution that make this possible: staging, language embedding and domain specific abstractions.

Staging means that one writes a program that creates another program that solves the original problem. In our case, the libraries we added to the Feldspar language implementation generate C code (with the help of the Feldspar core compiler) for bitvector and SIMD specific operations. This opens up more optimization possibilities compared to a library written in the object language.

Language embedding is a technique to implement a new (usually domain specific) language within an existing (usually general purpose) programming language. This technique is known to simplify DSL development, since there is no need to create lexer, parser and most of the semantic checks can also be done by the compiler of the so called *host language*. Furthermore, in case of the embedding technique [14] that Feldspar uses, the host language automatically becomes a meta language for the DSL. This means, in our case, that the full power of Haskell can be used to generate and combine Feldspar programs. This power also comes with safety: The rich and strict type system of Haskell is an essential in constructing the presented language abstractions. Recall that the most important difference between the presented unoptimized and optimized programs was their type, which was used to switch on or off the bitvector and SIMD specific optimizations.

Using *domain-specific abstractions* means that the language features do not aim to solve all kinds of programming problems, instead, they concentrate on a given problem domain and provide nicer and more efficient toolset for that particular domain. The solutions presented in this paper are good examples of this approach. *For loops* and indexing operations of C can be used to process or write an array in any particular order. The `map` , `fold` and other similar operations provided for Feldspar vectors and bitvectors are more restricted, but they are well-suited for the domain of Feldspar: digital signal processing. This means that the typical algorithms of this domain can be expressed more concisely and elegantly than using explicit loops an indexing. Furthermore, static analysis of arbitrary loops in order to achieve the optimizations presented in this paper is far more complex then our libraries. For example, to achieve the optimizations of the presented Bitvector or SIMD library using traditional compiler techniques one needs to unroll the loop, then change the order of instructions inside, finally find the instructions that can be collapsed into a single bitwise operation or platform-specific SIMD instruction. Note that the earlier mentioned *unroll* pragma, which is nontrivial in itself, is only the first step of this transformation.

The main message of the paper is that it is possible to combine the readability and maintainability of high level programming with advanced optimization techniques using proper domain-specific language elements. The embedded language Feldspar provided us with an environment where such language abstractions are possible to add modularly to the existing language. On the other hand, the ideas presented in this paper can also be used in other DSLs, if it is possible to add new abstract types, define operations on them and control how these are compiled to target code.

Acknowledgments. We are grateful to the grant EITKIC 12-1-2012-0001 that is supported by the Hungarian Government, managed by the National Development Agency, and financed by the Research and Technology Innovation Fund.

We also thank the anonymous reviewers for their comments which helped us improving this paper.

References

1. ISO/IEC 13211: Information technology Programming languages Prolog (1995)
2. Programming Languages – C, International Standard ISO/IEC 9899:201x (April 2011)
3. Ada Reference Manual, International Standard ISO/IEC 8652:2012(e) (2012)
4. The feldspar-language package in the Hackage library data base (2012), `http://hackage.haskell.org/package/feldspar-language`
5. Feldspar examples and generated target codes (2013), `http://feldspar.inf.elte.hu/ComSIS2013/`
6. The Glorious Glasgow Haskell Compilation System User's Guide, Version 7.6.3 (2013)
7. LangPop - Programming Language Popularity (2013), `http://langpop.com`
8. Tiobe Programming Community Index (2013), `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`
9. Axelsson, E., Claessen, K., Dévai, G., Horváth, Z., Keijzer, K., Lyckegård, B., Persson, A., Sheeran, M., Svenningsson, J., Vajda, A.: Feldspar: A domain specific language for digital signal processing algorithms. In: MEMOCODE. pp. 169–178. IEEE Computer Society (2010)
10. Chassaing, R., Reay, D.: Digital Signal Processing and Applications with the TMS320C6713 and TMS320C6416 DSK (Topics in Digital Signal Processing). Wiley-IEEE Press, 2nd edn. (2008)
11. Chiba, S.: Open c++ tutorial. Tokyo: University of Tsukuba (1998)
12. (ed.), S.M.: Haskell 2010 Language Report. URL http://www.haskell.org/onlinereport/haskell2010 (2010)
13. Franchetti, F., Püschel, M.: Generating SIMD vectorized permutations. In: International Conference on Compiler Construction (CC). Lecture Notes in Computer Science, vol. 4959, pp. 116–131. Springer (2008)
14. Hudak, P.: Modular domain specific languages and tools. In: Proceedings of the 5th International Conference on Software Reuse. pp. 134–142. ICSR '98, IEEE Computer Society, Washington, DC, USA (1998)
15. Kowarschik, M., Weiss, C.: An overview of cache optimization techniques and cache-aware numerical algorithms. In: Meyer, U., Sanders, P., Sibeyn, J. (eds.) Algorithms for Memory Hierarchies, Lecture Notes in Computer Science, vol. 2625, pp. 213–232. Springer Berlin / Heidelberg (2003)
16. McFarlin, D., Arbatov, V., Franchetti, F., Püschel, M.: Automatic SIMD vectorization of fast Fourier transforms for the larrabee and avx instruction sets. In: International Conference on Supercomputing (ICS) (2011)
17. Nuzman, D., Henderson, R.: Multi-platform auto-vectorization. In: Proceedings of the International Symposium on Code Generation and Optimization. pp. 281–294. CGO '06, IEEE Computer Society, Washington, DC, USA (2006)
18. Patterson, D.A., Hennessy, J.L.: Computer Organization and Design, Revised Fourth Edition: The Hardware/Software Interface. Elsevier, Inc., 225 Wyman Street, Waltham, MA 02451, USA, 4th edn. (2011)
19. Plauger, P., Lee, M., Musser, D., Stepanov, A.A.: C++ Standard Template Library. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edn. (2000)
20. Solar-Lezama, A., Rabbah, R., Bodík, R., Ebcioğlu, K.: Programming by sketching for bit-streaming programs. In: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation. pp. 281–294. PLDI '05, ACM, New York, NY, USA (2005)
21. Thielemann, H.: Compiling signal processing code embedded in haskell via llvm. In: CoRR. vol. 1004.4796 (2010)
22. Wu, P., Eichenberger, A.E., Wang, A.: Efficient simd code generation for runtime alignment and length conversion. Code Generation and Optimization, IEEE/ACM International Symposium on 0, 153–164 (2005)

Gergely Dévai is an assistant lecturer at Eötvös Loránd University, Faculty of Informatics, where he graduated from in 2004. From 2009 he also works for Ericsson Hungary, in the Software Technology group. His research interests are formal methods, language embedding and executable modeling.

Zoltán Gera graduated at Eötvös Loránd University, Faculty of Informatics at 2004. He started his research in digital signal processing there which is still his main area of interest. He is Senior Developer and Tech Lead at prezi.com.

Zoltán Kelemen is a student at Eötvös Loránd University, Faculty of Informatics. From 2013 he also works for Ericsson Hungary, as a software developer. His research interests are functional programming, language embedding, compilers, and concurrency.

Received: February 24, 2013; Accepted: July 25, 2014.