A Configurable Framework for Stream Programming Exploration in Baseband Applications

Jerker Bengtsson and Bertil Svensson

Centre for Research on Embedded Systems
Halmstad University
PO Box 823, SE-301 18 Halmstad, Sweden
{Jerker.Bengtsson, Bertil.Svensson}@ide.hh.se

Abstract

This paper presents a configurable framework to be used for rapid prototyping of stream based languages. The framework is based on a set of design patterns defining the elementary structure of a domain specific language for high-performance signal processing. A stream language prototype for baseband processing has been implemented using the framework. We introduce language constructs to efficiently handle dynamic reconfiguration of distributed processing parameters. It is also demonstrated how new language specific primitive data types and operators can be used to efficiently and machine independently express computations on bit-fields and data-parallel vectors. These types and operators yield code that is readable, compact and amenable to a stricter type checking than is common practice. They make it possible for a programmer to explicitly express parallelism to be exploited by a compiler. In short, they provide a programming style that is less error prone and has the potential to lead to more efficient implementations.

1. Introduction

Advanced embedded high-performance applications put very high requirements on computer systems design. Some examples are modern radar systems and baseband processing in radio base stations (RBS). Although the specific requirements are somewhat different, the computational characteristics are quite similar. Traditionally this kind of applications have required development of ASICs and special purpose hardware to cope with the requirements. Parallel architectures for high-performance applications has been a topic of research during many years. In recent years, results of this research and the advances in silicon process technology have opened up for a commodity market of highly parallel and reconfigurable architectures spanning from tens to several hundreds of processors on a single die [1, 2, 3].

Compiler technology and language development, on the other hand, have not kept pace with the advances in processor architecture. New approaches are required in order to exploit the vast amount of exposed parallelism and communication structures. On the one hand, languages must offer constructs and operations that allow a programmer to express parallelism and computations that are characteristic for a certain application domain. On the other hand, to enable efficient compilation, languages must be structured for a machine abstraction that correlates well with the target architectures. These arguments speak in favor of a domain specific approach rather than a general purpose programming approach.

The goal of our research is to investigate and develop a stringent programming and compilation framework for domain specific high-performance applications, targeting parallel and reconfigurable processors. In order to investigate what primitive language constructs, data types and operators are needed in an efficient programming language, our approach is to perform implementation experiments using realistic applications and experimental tools which can be used to quickly implement executable language prototypes. The application used for implementation studies in this work is baseband processing performed in 3G WCDMA radio base stations. An experimental framework has been implemented in Java to be able to perform quick prototype development and emulation of domain specific programming languages.

1-4244-0054-6/06/$20.00 ©2006 IEEE

This paper is organized as follows. A background and motivation for the work is given in Section 2. The configurable framework that has been developed for implementation experiments is presented in Section 3. In Section 4, the language *StreamBits* which has been implemented for baseband processing is presented. Section 5 shows experiments that were conducted to demonstrate the applicability of the language. Finally, the paper is summarized with conclusions and future work in Section 6.

2. Background

The baseband provides the modem functionality in a wireless communication system and constitutes the core in the 3G WCDMA technology. A radio base station provides a set of full duplex physical data and control channels, which are used to map higher layer data packets to physical radio frames. [4]. The baseband resources of an RBS are managed by a higher layer Radio Network Controller (RNC), which is responsible for traffic scheduling on the physical user channels provided by the baseband. The computations performed in the RBS mainly constitute data flows of bit-intensive protocol and signal processing, where the processing is controlled by service parameters given by the RNC. A baseband processing board is a complex unit, for which many design parameters have to be considered. Besides meeting the hard requirements in performance, it must provide scalability and be sustainable for evolutions in standards. At the same time, customers want low-cost RBS product solutions [5]. The life cycle of an RBS is measured in several decades, not years. To decrease initial product development costs it is an advantage if COTS components can be used to as large an extent as possible. Even if in-house ASIC solutions are hard to compete with in terms of performance and energy efficiency, they require large volumes in order to be a cost efficient solution. Also, considering the life cycle of an RBS, it is desirable not to encapsulate more functionality than necessary into ASICs at early product generations. New standard network functionalities are constantly released and it must be possible to incorporate these in existing platforms with minimal changes in hardware.

To meet these kinds of requirements in system design, the trend is that more of the baseband functionalities are implemented using programmable solutions. One approach to support hardware flexibility is to abstract the baseband implementation through definition of an application programming interface (API). Thus, the physical implementation of the baseband processing components can be disregarded by the programmer, and the functionality of the components can be implemented in either software or hardware.

2.1 Parallel and reconfigurable processors

Of specific interest for the addressed application domain are the array structured parallel and reconfigurable processors. Most of these architectures have been developed for the purpose of compute and data intensive applications such as baseband processing. In this paper the term processor is used for the entire parallel processor on a chip, whereas we refer to the constituent processing elements as PEs. This specific category of processors offers parallelism on different granularity levels, which provides a highly formable program mapping space. The PEs of the array are in general tightly coupled, using low-latency communication networks controlled by the instruction set. The exposed details of the low-latency interconnect structures, combined with the high degree of parallelism, makes it possible to enhance performance by arranging parallel computations as streams.

Parallel and reconfigurable architectures can be grouped into three categories after granularity and processing principle. The more coarse-grained are usually designed after the MIMD principle [6, 2]. The second category can be characterized as SIMD/vector machines, constituting clusters of vector or SIMD units, orchestrated by one or several single instruction stream controllers [7, 8]. Finally the third category are what can be called semi-static configurable arrays [3]. These are more fine-grained architectures; they resemble FPGAs, but the PEs are of word-level ALU type instead of bit-level type. What is common for these architectural categories is that they expose a lower than usual level of the hardware for configuration by the compiler.

A general principle for these architectures is that they have no hardware-implemented cache logic and that most are designed with distributed private memory. Thus, complex cache and coherence mechanisms can be removed in favor of more computation-oriented logic. Instead, the data access arrangement needs to be expressed and configured by the programmer and the compiler.

2.2 Stream programming and compilation

The flexibility and parallelism offered by parallel and reconfigurable architectures have increased the complexity for both the programmers and the compiler tools. Most current architectures are accompanied with a specific approach for programming and compiling, and many of these approaches are based

on the language C. This is done through either some machine specific extensions to the C syntax [8, 9] or as a combination with another machine specific language [2, 10]. This is not an optimal approach – neither from the perspective of application programming, nor for compilation efficiency. First of all, the C language and compilers have originally been developed and optimized for general purpose programming, to be compiled for a von Neumann based machine abstraction with a single instruction stream and global-memory abstraction. Thus, there is no support in the language to explicitly express application parallelism. Data-parallel operations usually have to be extracted from serial loop iterations [11]. The C language also imposes very liberal programming of memory usage, allowing global pointers, recursive function calls and dynamic memory allocation. Since the targeted parallel architectures have distributed memory and no automatic caching and coherence mechanisms, it is very hard – or perhaps even impossible – to produce performance efficient code based on this kind of programming.

One purpose with the baseband API is to be able to choose hardware components from several suppliers. Thus, the program code must be portable. Without a general programming language, which can be efficiently compiled to other architectures, a large amount of reimplementations would be required. For example, when a processor is programmed using an architecture specific combination of C (for function implementation) and a subset of VHDL (for the interconnect structures), porting programs to another architecture would most likely require changing programming paradigm as well as putting large efforts in code rewriting.

From the application point of view, one of the more interesting approaches taken is the stream programming paradigm. A stream program has the structure of a dataflow graph, constituting synchronous flows of data streaming through a pipelined set of functions [12, 9]. This is a natural way of expressing signal processing applications, which usually constitute pipelined execution of compute intensive filter kernels. Stream programming allows a programmer to explicitly express function parallelism and data locality in the program. The function dependencies in a stream program are limited to input and output streams between the functions in the flow graph; global data is not allowed to be expressed in a stream program. This deterministic flow description exposes information of function parallelism and data locality to a compiler.

One of the more interesting stream programming languages is StreamIt [12]. It is developed to be a portable programming language for array structured processors and, unlike other stream languages, both functions and program flow graphs are expressed using one language. The syntax is Java-like and it does not allow such things as global data allocation and function calls, as most C based languages do. The flow graphs are described using pipeline constructs, and functions are implemented as filters. An application can be expressed as an abstract component graph, using pipeline constructs, and the concrete API components can be defined by filter constructs. The filter construct provides a natural interface for autonomous stream functions in an API, which could as well be linked to a hardware defined component in the compiler process.

3. Experimental framework for stream programming

In this section we present a programming framework that has been developed for experiments with stream programming. Specifically, we are interested in experimenting with new language types and structures currently not supported in StreamIt. There are two important aspects that need to be addressed. First, the language must provide program structures that are natural to use for definition of abstract components. Second, it must offer primitive types and operators that allow a programmer to efficiently express application characteristic computations. With the experimental framework, it is possible to quickly set up programming experiments with StreamIt language extensions without the need for laboursome compiler modifications.

Our implementation studies of the WCDMA baseband standard show that the baseband functions require a high degree of low-level data manipulation on bit-fields, and also that many computations can be executed in data-parallel fashion [13]. When traditional high-level primitives, such as integers and bytes are used, low-level bit operations cannot be naturally expressed. Implementation of bit operations normally have to take machine-specific details, such as register word-length, into consideration. Furthermore, this kind of low-level programming is quite error prone and it would be desirable to perform compile-time type checks on such primitive operations.

With our framework it is possible to investigate how bit-level and data-parallel operations can be expressed more efficiently without considering machine-specific details when implementing algorithms. It is also possible to define type check rules. The framework is implemented in Java and it is based on a set of design patterns, which define the elementary structure of a stream language. This elementary structure is, to a large extent based, on the StreamIt language structure,

but it is not identical. New data primitives and stream constructs can rapidly be implemented by making extensions to the framework. In the next subsection we discuss the predefined basic language structures of the framework, highlighting new features not offered by the StreamIt language. Then we discuss the implementation of these structures in the framework. Finally, we discuss the implementation of primitive types and operators.

3.1 Basic language constructs

A stream program is constructed using `Filter` and `Pipeline` components, which form a network of functions and data streams, see Figure 1. The `Filter` is the basic construct in which instructions are grouped to perform a computation. The `Pipeline` construct is used to organize stream components into a composite network. A component is added to a pipeline by the `add(component)` command.

The `Filter` and `Pipeline` components are connected with I/O tapes which constitute the data streams flowing through the network. A tape is implemented by a FIFO buffer of homogeneous data types. In StreamIt, `Filter` and `Pipeline` components can only be attached to a single stream. In our framework, we support implementation of dual tapes – one for data streams and one for streams of configuration parameters. The data tape constitutes the stream on which a filter performs its computation. The configuration tape is used to stream reconfiguration parameters to filters throughout the distributed network.

A `Filter` has three execution modes – `init`, `work` and `configure`. Transitions between these modes are mapped automatically at compilation time and the programmer only needs to define the functionality within each mode. The `init` mode is executed once, before the first firing of the filter, to initialize variables and parameters. The `work` mode implements the computations performed when a filter is working in steady state. The `configure` mode is a new language feature not supported in StreamIt. It is executed once before each execution of the `work` mode. The `configure` mode has been implemented to support more flexible programming of parameter configuration of the baseband algorithms (recall the configuration parameters signalled from the RNC), which must be performed periodically during program execution. With a `configure` mode and a separate configuration tape, configuration programming can be defined and modified without any changes in the `work` mode.

As in StreamIt, the I/O streams are accessed by `peek()`, `pop()` and `push()` operators. The `pop()` operator reads and removes an item from the stream, while `peek()` reads the item but does not remove it from the stream. In our framework, `popD()`, `peekD()` and `pushD()` are used for data streams and `popC()`, `peekC()` and `pushC()` for configuration streams.

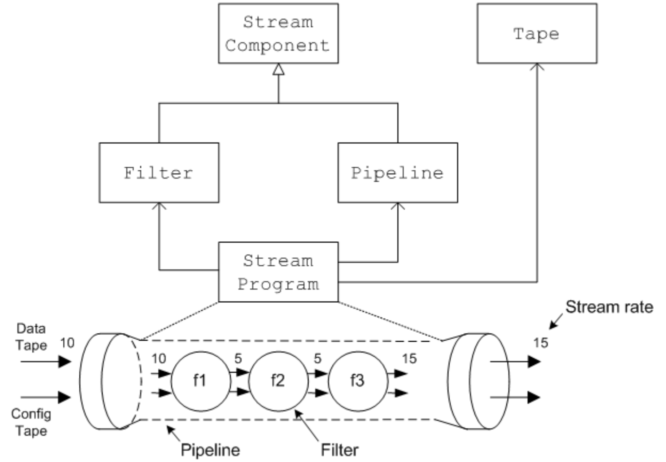

Figure 1. The framework structure

The framework is designed for stream programs with static stream rates only. That is, input and output stream rates must be defined constant by the programmer when implementing a filter. A reason for this restriction is to make it possible to, at compile time, check and assert stream rate compatibility between filters in the network.

3.2 Implementation of the basic language constructs

In this subsection we discuss how the basic language constructs are implemented in the Java-based framework. This includes `Tape`, `Filter` and `Pipeline` components, and it is shown how they are put together to form a stream network. The framework is structured using a set of design patterns in combination with type generics in Java 5.0 SDK [14].

The `StreamComponent` is a type-generic interface that defines the contractual functionality that a component must implement to be executable in a stream program. A `StreamComponent` must be attached to both data and configuration I/O tapes, where each tape can be a stream of different data type. This is handled elegantly by usage of generics in Java. The interface is parameterized using four generic types, for data and configuration streams respectively. The generic data types are instantiated by the class that implements the interface.

There are currently two basic components in the

framework which implement the `StreamComponent` interface – `Filter` and `Pipeline`. The `Filter` is a generic component that defines the basic structure of a filter construct in a stream language. The abstract parts of the `Filter` component constitute methods for `work`, `init` and `configure`, which must be implemented by a programmer to define the execution in these modes. The `configure` and `work` modes are called automatically in a deterministic order.

The `Pipeline` defines how `StreamComponents` are ordered to form a stream subnetwork. Since a `Pipeline` is a `StreamComponent` itself, it is possible to construct hierarchical pipelines. When a `StreamComponent` is attached to a pipeline with the `add(component)` operation, the stream rate compatibility with the preceding and the following components is checked. Both the `Pipeline` and the `Filter` templates are defined with type-generics for input and output streams. These types are defined by the programmer when instantiating a `Filter` or `Pipeline` using the `Filter` and `Pipeline` component templates.

Tapes are defined by a generic `Tape` component. The buffer data type is defined when a programmer makes an `add(component)` operation. The buffer size is determined at compilation time using the I/O stream rate directives that must be specified by the stream programmer when instantiating a component using `Pipeline` and `Filter` component templates.

The `StreamProgram` component is the top-level pipeline in a stream program. The programmer adds components to the program by using the `add(component)` directive in the method `streamProgram`, which is the main function called automatically at program execution. Besides adding components to the main pipeline, the programmer must also define the I/O stream types for both data and configuration streams.

3.3 Stream data types and operators

We now discuss the implementation of components for type definition and type operators. One of the main goals with the framework is to be able to elaborate with primitive stream data types for baseband applications. The framework allows strong type checking definitions on operations with primitive types. Since the framework is implemented using Java, some of the type checking must be done during run-time. However, in a real compiler implementation these type checks would be performed at compile time.

The `StreamType` is a generic interface for implementation of stream data primitives. This interface defines a common subset of abstract arithmetic, logic, relational and typecast operators. The `StreamType` interface must be implemented when defining a new stream data type, which in turn requires the common operations to be defined by the implementing type. Operators defined by `StreamType` (that are not valueless) take a generic type as input and return a value of generic type.

A major strength is that type checking rules can be defined for each data type that implements the `StreamType` pattern. All primitive stream types are implemented as abstract data types in Java.

4. Implementation of StreamBits

The framework has been used to implement *StreamBits*, which is a prototype language for baseband API development. The baseband input consists of bit-serial data streams that must be processed within hard real-time intervals. Currently, we have focused on the primitive type system and operators suitable for efficient implementation of bit-field manipulations and data-parallel operations. In this section we present these new types and operators. To demonstrate the advantages with our approach, we compare these types and operators with C-based expressions.

When a traditional C-programming approach is used, computations on bit-streams require a large amount of assembly-like machine-dependent expressions based on bit masks and shifts. This normally results in source code that is very hard to read and understand. Also, this kind of operations requires careful implementation by the programmer, since C-like languages lack type notions for bit field computations and therefore can not provide type safety. Furthermore, it results in machine dependent code, since the programmer must pack bits and calculate masks and shifts that are bound to fixed-length machine registers.

StreamBits has been implemented with types for bit-field and data-parallel operations. The definitions of these types are presented in Table 1. The *StreamBits* primitives are listed in the first column of the table and the corresponding type expressions in C primitives in the second.

Types for bit-fields. `bitvecST` is the type for declaration of bit-fields of length n. Thus, it is possible to define a set of n distinct bit-field types $t(n)$. In comparison, since in C, there is no primitive type notion for bit-fields, such data quantities must be expressed using integer or byte types. Therefore, type-correct operations on integers are also type-correct for bit-fields of arbitrary length. This type-mismatched bit mapping is quite error prone and not desirable.

Data-parallel type. `vecST` is a type that allow

Table 1. StreamBits types compared with C

StreamBits type	C type
$bitvecST(n)$ is a bit field type for each value of n	*int*, *byte*
$vecST(e_{0w}, e_{1w}, e_{2w}, e_{3w})$ is a type for parallel vectors with elements e_n of width w	$int[4]$, $byte[4]$ scalar array of *int* or *byte*

Table 2. StreamBits compared with C

bitvecST oper.	Corresponding C expr.
$bitslice(m:n)$	$(t\ \&\ w_{m:0})$
$bitsliceL(m:n)$	$(t\ \&\ w_{m:0}) \ll (w-m)$
$bitsliceR(m:n)$	$(t\ \&\ w_{m:0}) \gg n$
$bitslicePack(m:n)$	N/A
$lmerge(k:l, m:n)$	if $l <= (m-n)$: $((t\ \&\ w_{k:l}) \ll C_1) \mid ((s\ \&\ w_{m:n}) \gg n)$ if $l > (m-n)$: $((t\ \&\ w_{k:l}) \gg C_2) \mid ((s\ \&\ w_{m:n}) \gg n)$

fine-grained data-parallel operations to be expressed explicitly within a `Filter`, see row 2 in Table 1. The `vecST` type is defined as a vector of four elements, each of 32-bit width. Note that the definition of the vector type is parameterized by the number of elements e_n and the bit-field width w. Since there is no parallel notation in C, vector data are usually expressed using array constructs which are accessed scalar-wise.

Bit-field operations. A sub-field in `bitvecST` types is accessed using `bitslice` operators. Bit-fields of `bitvecST` type can also be merged using the `lmerge` operator. In Table 2, we list these operators and, for comparison, the corresponding C expressions. Bit-field upper and lower boundaries are annotated with k, m and l, n respectively. w is used for machine word-length, and bit masks of machine register length are annotated with $w_{m:0}$, where m represents the upper boundary of the bit mask.

The `bitslice` operator is currently defined for two cases – unaligned and aligned bit-slicing. Unaligned bit-slicing is performed with the `bitslice("m:n")` operator, row 1 in the table. The operator `bitslice("m:n")` produces a `bitvecST` with the same length as the operand, where bits m through n are copied from the corresponding bit-field in the operand, and the rest are set to 0.

Aligned bit-slicing is performed by the operators `bitsliceL("m:n")`, `bitsliceR("m:n")` and `bitslicePack("m:n")` listed on rows 2 through 4. The `bitsliceL("m:n")` operator produces a bit-field copy, like `bitslice("m:n")`, but the copied bit-field is aligned to the `bitvecST(`n`)` upper bound n. Similarly, the `bitsliceR("m:n")` operator produces a bit-field copy aligned to the lower bound. The corresponding expressions in C, on rows 2 and 3, require logical AND and a $SHIFT$ operations. In comparison, the `bitsliceL` and `bitsliceR` operators allow this operation to be more compactly expressed than the required C expression. Also, in the C expression for left-aligned masking on row 2, it is assumed that the upper bound is equal to the word length. But, since the `bitvecST(`n`)` type is defined for bit-fields of length n, this is generally not the case. The alignment of `bitsliceL` and `bitsliceR` values in *StreamBits* can be handled automatically at compile time.

The `bitslicePack` produces a left-aligned bit-field copy, just like the `bitsliceL` operator, but the result is packed into a `bitvecST(`s`)` where the field length s is equal to the length of the copied bit-field $(m-n)$.

The `lmerge` operator is used to merge two `bitvecST` bit-fields. The result is a `bitvecST(`s`)` where the length s is the sum of the two merged bit-field lengths $(k-l+m-n)$. The first operand is aligned to the left of the second operand. A right-aligned merge can be achieved by simply switching the order of the operands. The corresponding C expression requires at most 5 operations. The first operand should be aligned to the left of the second, which requires either a left or a right shift, depending on if the masked bit-fields are overlapping or not ($l <= (m-n)$ or $l > (m-n)$). This alignment is a shift constant specified by the programmer (C_1 and C_2). Thus, the if cases are used only to mark two separate alignment cases.

Data-parallel operations. Besides the basic arithmetic and logical operations, the `vecST` type also supports bit-field operations, such as `lmerge`, `vecslice`, `vecsliceL` and `vecsliceR`, see Table 3. The `lmerge` operation is defined precisely as `lmerge` for `bitvecST` types. The merge is performed *in parallel* for each of the vector elements and the result is of type `vecST`. In *StreamBits*, the maximum length of an element merge is w.

The `vecslice(m:n)`, `vecsliceL(m:n)` and `vecsliceR(m:n)` operators are vector-parallel versions of `bitslice` operators. Since there is no corresponding parallel notation in C, parallelism must be transformed into sequential expressions, typically using loop constructs and scalar data arrays. This is illustrated in the right part of Table 3. Few will argue that it is a natural way of expressing application parallelism – to use sequential scalar constructs, which are then to be parallelized by a compiler that is only aware of the sequential constructs given by the programmer.

Some computations require scalar processing of `vecST` elements. Scalar elements in `vecST` are accessed

Table 3. Operator comparison vecST

StreamBits oper.	Corresponding C expression
$vecslice(m:n)$	$for\ i = 0\ to\ 4\{t[e_i]\ \&\ w_{m:n}\}$
$vecsliceL(m:n)$	$for\ i = 0\ to\ 4\{t[e_i]\ \&\ w_{m:n} \ll (w-m)\}$
$vecsliceR(m:n)$	$for\ i = 0\ to\ 4\{t[e_i]\ \&\ w_{m:n} \gg n\}$
$lmerge(k:l, m:n)$	$for\ i = 0\ to\ 4\{$ if $l <= (m-n)$: $(t[e_i]\ \&\ w_{k:l}) \ll C_1 \mid (s[e_i]\ \&\ w_{m:0}) \gg n$ if $l > (m-n)$: $(t[e_i]\ \&\ w_{k:l}) \gg C_2 \mid (s[e_i]\ \&\ w_{m:0}) \gg n$

by `getElement(`e_i`)` and `setElement(`e_i`,` val`)` operators, where e_i is the element index of the vector and val is the value.

5. Experiments with 3G UMTS baseband functions

In order to demonstrate and evaluate the applicability of the *StreamBits* language, we have conducted experiments with baseband functions. In this paper two different implementation examples of one baseband function are presented – cyclic redundancy check (CRC) processing for a voice call service and for high bit-rate data services [13].

CRC processing for voice services. Voice calls are coded using an adaptive multi rate codec (AMR). The baseband data input constitutes three AMR encoded bit-streams mapped onto separate transport channels, A, B and C. Each stream constitutes coded speech data of different importance to the quality of a voice channel; the A bits are the most important and the C bits the least. The channels are processed with different baseband parameters; only the A channel bits are transmitted with a CRC computed checksum. The CRC implementation, shown in figure 2, is a table-driven algorithm which encodes a single transport channel (Note that this is a selected part and not the complete CRC baseband function). The variables, except for the loop counter, are all of `bitvecST` type. In each loop iteration, this algorithm encodes eight-bit long fields of the input stream. The input bits are packed into a stream of `bitvecST` data type. Each `bitvecST` is 32 bits long, thus four iterations are required to process each `bitvecST` that is read from the input stream. Defined by the code within the `if` clause, every fourth iteration a new `bitvecST` value is read into a temporary input register (`r_tmp`), while the previous one is pushed to the output stream.

On line 7, the next 8-bit field to be encoded is read from the encoder register `r` using the `bitsliceR` operator. On line 8, the MSB from the temporary input register `r_tmp` is shifted into the LSB of the encoder register `r` using `bitslice` and `lmerge`.

```
for(intST cnt;cnt.lt(len);cnt.Assign(cnt.incr())){
1.  if(cnt.mod(new intST(4)).eq(new intST(0))){
2.      out.Assign(temp);
3.      temp.Assign(peekD());
4.      r_tmp.Assign(temp);
5.      pushD(popD());
6.  }
7.  t.Assign(r.bitsliceR("31:24"));
8.  r.Assign(r.lmerge("23:0", r_tmp.bitsliceR("31:24")));
9.  r.Assign(r.XOR(table[crc_poly.getVal()][t.getVal()]));
10. r_tmp = r_tmp.lshift(8);
}
```

Figure 2. CRC implemented in StreamBits

Finally, the lookup value is read from the table to be XOR:ed (the division), with the encoder register, and the next 8-bit input in the temporary register is aligned to the MSB position for the next iteration.

CRC processing for high bit-rate data. Data transmissions can be mapped using multiple transport blocks of equal size, mapped on multiple transport channels. Since there are no data dependencies between the transport blocks, multiple blocks can be processed in parallel using SIMD control.

The *StreamBits* code in Figure 3 represents a SIMD parallel implementation of the same encoding algorithm previously demonstrated for the AMR service. The `cnt` and `v0-v3` variables are scalar variables of `intST` type, and the other are of `vecST` type. Like in the AMR example, the parallel encoder encodes 8 bits of the input stream per iteration.

Each data item in the I/O streams constitutes a 32-bit field of four simultaneous transport block input streams; one transport block stream per vector element. The parallel `vecslice` operator on line 7 copies the next 8 bits of each vector element in the encoder register `r`, and stores them aligned with the lsb positions in the `t` vector. Each element in `t` constitutes the next lookup index for each bit-stream being processed. On line 8, new input bits are shifted into the encoding register by copying the MSB of the input vector register `r_tmp`, which are merged with the remaining bits in the encoder register. Like for the `bitslice` operators, no machine dependent masking and shifting needs to be expressed in the code.

The table look-ups performed on lines 9 through 12 have to be expressed with scalar operations; one look-up for each element. This is because the input bit-fields in register `t` constitute arbitrary values from the four transport blocks and therefore it is not possible to vectorize the look-up operation. However, this does not mean that this portion of the code cannot be executed in parallel. Finally, the scalar values with the look-up values are vectorized and XOR:ed with the encoder

```
for(intST cnt;cnt.lt(len);cnt.Assign(cnt.incr())){
1:  if(cnt.mod(new intST(4)).eq(new intST(0))){
2:    out.Assign(temp);
3:    temp.Assign(peekD());
4:    r_tmp.Assign(temp);
5:    pushD(popD());
6:  }
7:  t.Assign(r.vecslice("31:24"));
8:  r.Assign(r.lmerge("23:0", r_tmp, "31:24"));
9:  v0.Assign(table[poly.val()][t.getElement(0).val()]);
10: v1.Assign(table[poly.val()][t.getElement(1).val()]);
11: v2.Assign(table[poly.val()][t.getElement(2).val()]);
12: v3.Assign(table[poly.val()][t.getElement(3).val()]);
13: r.Assign(r.XOR(new vecST(v0, v1, v2, v3)));
14: r_tmp.Assign(r_tmp.lshift(8));
}
```

Figure 3. CRC processing for DATA

register on line 13, and the next input bits to be shifted into the encoder register are shifted to the msb positions in the temporary input register, `r_tmp`.

6 Conclusions and future work

We have described a configurable framework to be used for experiments with stream programming development targeting embedded high-performance applications, such as baseband processing in radio base stations. A domain specific language prototype for baseband processing, called *StreamBits*, was implemented to demonstrate the use of the framework and to perform implementation experiments. We introduced stream constructs to be able to efficiently program dynamic reconfiguration of distributed processing parameters. It was shown that the language has the potential to lead to more compact and efficient codes for bit-field and data-parallel computations, compared to when typical von Neumann based languages, such as C are used. The primitive types in the language impose a programmer to explicitly express functions with inherent, fine-grained data parallelism. Moreover, it was demonstrated how the primitive data-parallel vector and bit-vector data types and operations can be used without exposing machine specific details such as register word lengths. Another advantage with the introduced primitive data types is the opportunity to perform strong type checking on low-level bit operations.

Future work will be focused on the definition of a tailored stream processing language for baseband API development. This will be based on extended prototype experiments using the configurable framework. The development will include the implementation of a compiler framework for a parallel machine abstraction, which can be applied for efficient mapping on parallel and reconfigurable, array structured processors. To support efficiency, the language should offer parallel expressions. To support portability, the syntax should not allow machine-specific details in the code.

Acknowledgment

The authors would like to thank Dr. Anders Wass at Ericsson AB and Dr. Veronica Gaspes at Halmstad University for valuable advice and suggestions. This work has been funded by research grants from Ericsson AB and the Knowledge Foundation.

References

[1] Freescale Semiconductor. MRC6011: Reconfigurable Compute Fabric (RCF) Device. www.freescale.com, Oct. 2004.

[2] G. Panesar A. Duller and D. Towner. Parallel Processing - the picoChip way! In *Proc. of Communicating Process Architectures*, pages 125–138, 2003.

[3] PACT XPP Technologies. XPP-IIb Core Overview. www.pactcorp.com, Sept. 2005.

[4] H. Holma and A. Toskala. *WCDMA for UMTS: Radio Access for Third Generation Mobile Communications.* Addison-Wesley, third edition, 2004.

[5] J. Lerzer Z. Zhang, F. Heiser and H. Leuschner. Advanced baseband technology in third-generation radio base stations. *Ericsson Review*, (01):32–41, 2003.

[6] M. B. Taylor et al. Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams. In *Proc. of Int. Symposium on Computer Architecture*, pages 2–13, 2004.

[7] J. H. Ahn et al. Evaluating the Imagine Stream Architecture. In *Proc. of Int. Symposium on Computer Architecture*, pages 14–25, 2004.

[8] A. E. Eichenberger et al. Optimizing Compiler for a CELL Processor. In *Proc. of Int. Conf. on Parallel Architectures and Compilation Techniques*, pages 161–172, 2005.

[9] A. Das et al. Imagine Programming System User's Guide. www.cva.stanford.edu/imagine, April 2004.

[10] PACT XPP Technologies. Programming XPP-IIb Systems. www.pactcorp.com, Sept. 2005.

[11] S. V. Rajopadhye. Dependence Analysis and Parallelizing Transformations. In *The Compiler Design Handbook*, pages 329–372. CRC Press, 2002.

[12] M. I. Gordon et al. A Stream Compiler for Communication-Exposed Architectures. In *Proc. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 291–303, 2002.

[13] J. Bengtsson. Baseband Processing in 3G UMTS Radio Base Stations. Technical Report IDE0629, Halmstad University, 2006.

[14] E. Gamma. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series. Addison-Wesley, 1995.