Native code compilation of Erlang's bit syntax

Per Gustafsson
Computing Science Dept.
Uppsala University, Sweden
pegu2945@csd.uu.se

Konstantinos Sagonas
Computing Science Dept.
Uppsala University, Sweden
kostis@csd.uu.se

ABSTRACT

ERLANG's bit syntax caters for flexible pattern matching on bit streams (objects known as *binaries*). Binaries are nowadays heavily used in typical ERLANG applications such as protocol programming, which in turn has created a need for efficient support of the basic operations on binaries.

To this effect, we describe a scheme for efficient native code compilation of ERLANG's bit syntax. The scheme relies on *partial translation* for avoiding code explosion, and improves the performance of programs manipulating binaries by translating frequently occurring instances of BEAM instructions into native code via an intermediate translation to instructions of a register transfer language. Our performance evaluation shows that in a HiPE-enabled Erlang/OTP system, the obtained speedups are often significant.

1. INTRODUCTION

Functional programming languages traditionally manipulate objects such as numbers (integers and floats), atoms (sequences of alphanumeric constants), lists and structures. Some of them also provide a notation for records that allows abstraction and often (some form of) object oriented-style program development. ERLANG supports all these types of objects, but also includes a datatype typically not found in other functional languages: *binaries*.

It should be mentioned that the binary datatype was added to ERLANG to cater for application needs: Binaries were first introduced into ERLANG in 1992 to provide an efficient container for object code. Subsequently, it was recognized that binaries can be used in applications that perform extensive I/O, networking TCP/IP-style of I/O, in GUI systems, and most importantly in protocol programming typically developed by telecommunication applications or other uses of ERLANG. Recognizing the importance of binaries, in 1999, a proposal for a binary datatype was presented in [4]. This datatype was more powerful than the original binary, since it was introduced together with a new bit syntax which made it possible to easily build and match binaries. In 2000, a revised version of the datatype, presented in [2], was introduced into the Erlang/OTP system.

Erlang Workshop '02 Pittsburgh, USA

Since then, binaries have been used extensively both in various libraries of Erlang/OTP and in user applications. Indeed, nowadays the performance of many ERLANG programs crucially depends on the efficiency of the underlying operations that support manipulation of binaries. The BEAM, the virtual machine of Erlang/OTP, maps these operations to virtual machine instructions implemented directly in C. As a result, their performance is quite good. Till recently, the HiPE native code compiler [1, 3], did not handle these binary instructions specially, but instead treated them as built-in functions implemented through straightforward calls to the corresponding BEAM C functions. As a result, the performance of manipulating binaries in HiPE was more or less on a par with the BEAM-based system and actually slightly disadvantaged due to the mode-switch overhead that is involved in calling a C function of the BEAM interpreter from native code. Wanting to improve this situation, we have embarked on a project to directly compile ERLANG binaries to native code. The approach we took and the compilation scheme we follow are described in this paper. Besides documenting our implementation, which will be included in the upcoming Erlang/OTP Release 9, we shed more light on the internals of ERLANG binaries whose implementation is not described anywhere, and we hope that such information is potentially useful to other functional programming implementors that consider adding binaries to their language of choice.

The rest of the paper is structured as follows. The next section begins by reviewing ERLANG's binary data object and the bit syntax that supports the flexible manipulation of binaries. We then (Section 3) describe how the operations that ERLANG's bit syntax allows for are translated to appropriate virtual machine instructions. We focus attention on the instructions that efficiently support matching. In Section 4, we present a translation of these virtual machine instructions to a register transfer language (RTL). This translation is made in two steps: First the virtual machine code is translated into an intermediate code representation, and this intermediate code is translated to RTL code. Section 5 discusses a scheme to *a priori* calculate the heap space a binary matching operation requires. We end the paper by evaluating the performance of our implementation both on ERLANG protocol programming applications and on synthetic benchmarks, and by quantifying how parameters such as byte alignment influence the efficiency of the generated code.

2. BINARIES

The binary datatype represents a stream of bits of a size that is evenly divisible by 8. Two basic operations can be performed with the binary: *creation* of a new binary and *matching* against an existing binary.

2.1 Bit syntax

A bit syntax expression (called a Bin in [2]) is what allows ERLANG users to construct binaries and match binary patterns. A Bin is written with the following syntax:

$$\texttt{<<Segment}_1\texttt{, Segment}_2\texttt{, ..., Segment}_n\texttt{>>}$$

The Bin represents a low level sequence of bytes. Each of the $\texttt{Segment}_i$'s specifies a *segment* of the binary. A segment represents an arbitrary number of contiguous bits in the Bin. The segments are placed next to each other in the same order as they appear in the bit syntax expression.

2.1.1 Segments

Each segment expression has the general syntax:

```
Value:Size/SpecifierList
```

where both the `Size` and the `SpecifierList` can be omitted since there are default values for these specifications. The `Value` field must however always be specified. In a binary match, the `Value` can either be an Erlang term, a bound variable, an unbound variable, or the wildcard variable '_'. The `Size` field can either be an integer or a variable that is bound to an integer. The `SpecifierList` is a dash-separated list of up to four options that specify type, signedness, endianism, and unit. The different forms of type specifiers are described in Table 1. If all of these possibilities are used, the syntax of each segment expression is:

```
Value:Size/Type-Signedness-Endianism-unit:Unit
```

The `Size` specifier gives the size of the segment measured in units. Thus the size of the segment in bits (hereafter called its *effective size*) will be $\texttt{Size} * \texttt{Unit}$.

2.1.2 Binary Matching

This is the syntax for matching with a binary if `Binary` is a variable bound to a binary:

$$\texttt{<<Segment}_1\texttt{, Segment}_2\texttt{, ..., Segment}_n\texttt{>> = Binary}$$

The $\texttt{Value}_i$ fields of the $\texttt{Segment}_i$ expressions that describe each segment will be matched to the corresponding segment in `Binary`. For example, if the $\texttt{Value}_1$ field in $\texttt{Segment}_1$ contains an unbound variable and the effective size of this segment is 16, this variable will be bound to the first 16 bits of `Binary`. How these bits will be interpreted is governed by the `SpecifierList` of $\texttt{Segment}_1$.

Example 2.1 As shown below, binaries are generally displayed as a sequence of comma-separated unsigned 8 bit integers inside `<<>>`'s. The ERLANG code:

```
Binary = <<10, 11, 12>>,
<<A:8, B/binary>> = Binary
```

results in the binding `A = 10, B = <<11, 12>>`.

Here `A` matches the first 8 bits of `Binary`. Because of the default values, these eight bits are interpreted as an unsigned, big-endian integer. `B` is matched to the rest of the bits of `Binary`. These bits are interpreted as a binary since that type specifier has been chosen. Because of that, `B` matches to the rest of `Binary`, as this is the default size for the `binary` type specifier.

Matching against a binary can be used in a function head or in an ERLANG `case` statement just like any other matching operation.

Example 2.2 Consider the code shown below:

```
case Binary of
    <<13:8/integer, X/binary>> -> X;
    <<_:8, X:16/integer, _/binary>> -> X;
    <<X/binary>> -> error
end.
```

Here `Binary` will match the pattern in the first branch of the case statement if its first 8 bits represented as an unsigned integer have the value of 13. In this case, `X` becomes a binary consisting of the rest of the bits of `Binary` and is returned by this statement. If this is not the case, then `Binary` will match the second pattern if `Binary` is larger than three bytes, since there are no demands on the values of what is matched out from the binary in the pattern of the second branch. `X` will be bound to an integer consisting of bit 9 to bit 24 in `Binary`. These bits will be taken in big-endian order and will be interpreted as an unsigned integer. If the binary does not match either pattern of the two first branches, `X` will be matched to `Binary` and the atom `'error'` will be returned. Three examples of matchings using this code are shown below.

Binary	X matches to
<<13, 14, 15>>	<<14, 15>>
<<12, 1, 2, 20>>	258
<<0, 255>>	<<0, 255>>

Types. The bit syntax allows three different types to be specified: integers, floats and binaries. The `integer` type is the default type and can be of any size. For integers, the user can also specify endianism and signedness; see below. The default specifications for an integer are `Size` of 8, unsigned, big-endian, and a unit of 1.

The `float` type only allows effective sizes of 32 or 64 bits. The user can also specify endianism. The default specifications for a float are a size of 64, in big-endian format, and with a unit of 1.

The `binary` type allows effective sizes that are evenly divisible by 8. Specifying endianism or signedness does not modify how a binary is matched. The default specifications for a binary is the size `all` which means the binary is being matched out completely. If a size is given by the user the default unit is 8.

Example 2.3 Below we show some binaries and their default expansions.

binary	by default expands to
<<X>>	<<X:8/integer-unsigned-big-unit:1>>
<<X/float>>	<<X:64/float-big-unit:1>>
<<X/binary>>	<<X:all/binary>>
<<X:Size/binary>>	<<X:Size/binary-unit:8>>

Table 1: Binary type specifiers.

`integer`	The bit stream will be interpreted as an integer. This is the default setting.
`float`	The bit stream will be interpreted as a float. It can only have the sizes 32 and 64.
`binary`	The bit stream will not be interpreted. The default `unit` size of a binary is 8.
`signed`	The bit stream will be interpreted as an integer in 2's complement sign extension (only applies to integers).
`unsigned`	The bit stream will be interpreted as an unsigned integer. This is the default setting.
`big`	The bytes will be picked in big-endian order. This applies to floats and integers and is the default setting.
`little`	The bytes will be picked in little-endian order. Applies to floats and integers.
`unit`	Typically followed by '`:`' and an integer between 1 and 256. The integer is multiplied with the size term to produce the *effective size*. This is typically used to ensure either byte-alignment in a binary match or that a new binary has a size that is divisible by 8 regardless of the values of the runtime-variables. The default setting is 1 for integers and floats and 8 for binaries.

Endianism. An endianism specifier is used to decide in which order bytes should be picked when forming an integer or a float. `big` means that the bytes are picked in big-endian order, while `little` means the bytes are picked in little-endian order.

Example 2.4 If `X` and `Y` are unbound variables, the matching:

```
<<X:16/integer-big>> = <<0, 42>>
```

results in the binding `X = 42` as the eight low bits of `X` are 42, while the matching:

```
<<Y:16/integer-little>> = <<0, 42>>
```

results in the binding `Y = 10752` (i.e., 42 * 256) since 42 now appears in the eight high bits.

Signedness. A signedness specifier is introduced to allow matching of either signed or unsigned integers. The default value is `unsigned`. This means that the bit stream is interpreted as an unsigned integer. The `signed` specifier makes sure that the bit stream is interpreted as an integer with a two complement sign extension. We note that the `signed` and `unsigned` specifiers are actually allowed in all expressions, but they only have a meaning when matching integers.

Example 2.5 If `X` and `Y` are unbound variables, the code:

```
<<X:8/integer-unsigned>> = <<255>>,
<<Y:8/integer-signed>> = <<255>>
```

results in the binding `X = 255, Y = -1`.

Tail of a binary. In Example 2.2, the last segment of each of the three binaries in the patterns of the `case` statement shows an interesting use of the `binary` type specifier without a size value in matchings. In a sense, this use is similar to the familiar list-tail operator since it will let the size become the size of the rest of the binary that is matched out. It is however important to know that a binary can only have a bit-size that is evenly divisible by eight. This also applies when the wildcard variable is used as `Value`.

2.2 Internal representation

Although to the user binaries appear as only one type, the current internal representation of binaries in Erlang/OTP includes three different types of binaries: *heap binaries*, *reference counted binaries* (REFC binaries), and *sub-binaries*.

Heap binary is a variable-sized object containing a type tag, a size field, and the data contained in the binary. As its name implies, this type of binary objects are stored on the heap. This representation is used for small-sized binaries (i.e., binaries smaller than a particular constant MAX_HEAP_BIN_SIZE). In the current process-centric heap architecture of Erlang/OTP, heap binaries can be costly to send as a message since the whole binary is being copied to the heap of the receiving process.

Reference counted binary is an object that points to an off-heap reference counted data structure. This data structure contains all the binaries. All REFC binaries on a process' heap are linked together in a list and after garbage collection the list is traversed. When a process holding a pointer to a REFC binary dies, the reference counter is decreased. When the counter reaches zero, the binary structure can be deallocated. On the other hand, this counter is increased when a REFC binary is sent as a message. This type is used for large binaries and is effective since it becomes extremely cheap to send as a message between processes. This is because only the header that resides in the heap of the sender needs to be copied to the heap of the receiver process.

Sub-binary is a heap object that is a pointer to another binary object with an offset and a size field that describe how far into the other binary this binary starts and how many bytes of data it consists of, respectively. This representation is useful since it allows to cheaply split a binary into several smaller binaries. The cost of sending a sub-binary as part of a message depends on the type of binary that the sub-binary points to. When it points to a heap binary, then the whole heap binary has to be copied to the other process as well as the sub-binary. When the sub-binary points to a REFC binary however only the sub-binary and the REFC binary header have to be copied.

3. VIRTUAL MACHINE TRANSLATION

An intermediate step towards efficient compilation of the bit syntax is to make a translation to appropriate virtual machine instructions. This can be done by translating the different bit syntax expressions to corresponding operations of the virtual machine. Indeed, the BEAM (the virtual machine of Erlang/OTP R8B) follows this approach. The type specifiers other than the `unit` specifier are encoded by one bit each in a Flags argument. The second bit is set if the

operation is little-endian and the third bit is set if the operation is signed. To simplify future steps in the compilation process, a third flag shows whether the operation is guaranteed to start at a byte boundary. When this is the case the first bit is set. The Flags argument will thus be an integer between 0 and 7. The Size argument will be taken directly from the size field in the bit syntax expression. It will either contain an integer, a variable, or the atom `all`. As mentioned, the `all` atom is only possible when the `binary` type specifier is involved.

3.1 Binary matching

The matching operation can be translated into eight different BEAM instructions. One instruction to initiate the matching, another to test if a match has ended successfully, two instructions to save and load the state of the matching, one instruction to skip forward in the binary, and finally three instructions to read bits from the binary. There are also some BEAM instructions that are not specific to binaries, but are nevertheless used within the matching code. For example, the is_equal_exact or switch-type instructions. In this paper we do not concern ourselves with non binary-specific instructions.

Since this is a matching operation, the matching flow will be in standard matching form. This means that the matching operations have a SuccessLabel and a FailLabel. When the operation succeeds the flow continues at the SuccessLabel and when it fails it continues at the FailLabel. A match is over either when a bs_test_tail instruction has succeeded, or when the flow has reached the FailLabel associated with bs_start_match. In the BEAM, the state of a binary matching is maintained in 4 global variables shown in the table below.

BinBase	the address to the first byte in the binary that is used in the matching
BinSize	the size in bits of the binary that is matched
BinOffset	number of bits that have been matched so far
BinOrig	the ERLANG term of the REFC or heap binary that points to the binary being matched

The BEAM instructions used for binary matching are:

void bs_start_match(binary Bin)

This BEAM instruction is used when encountering the `<<` operator in a `<<Segment`$_1$`, ..., Segment`$_n$`>> = Bin` matching expression. It initializes the state variables from the information contained in `Bin`.

Fail Conditions

- Wrong argument type

void bs_skip_bits(uint Size, unit Unit, uint Flags)

This BEAM instruction is used for the translation of a `_:Size/`*type*`-unit:Unit` expression. Recall that if the *type* is `binary`, `Size` does not have to be an unsigned integer but it can also be the atom `all`. This operation will increase *BinOffset* by `Size` * `Unit` or if `Size` is the atom `all` it will set *BinOffset* to *BinSize*.

Fail Conditions

- (*BinOffset* + Size ∗ Unit) > *BinSize*
- Size = `all` and *BinOffset* is not evenly divisible by 8
- Wrong argument type

int bs_get_integer(uint Size, uint Flags, uint Unit)

This BEAM instruction is used for the translation of a `Int:Size/integer-`*specifier*`-unit:Unit` expression. It will load the first Size ∗ Unit bits from the position *BinBase* + *BinOffset* and turn these bits into an integer. The Flags argument specifies if the bits will be interpreted as representing a signed or unsigned integer and if the bytes will be taken in big- or little-endian order.

Fail Conditions

- (*BinOffset* + `Size` ∗ `Unit`) > *BinSize*
- Wrong argument type

float bs_get_float(uint Size, uint Flags, uint Unit)

This BEAM instruction is used for the translation of a `Float:Size/float-`*specifier*`-unit:Unit` expression. It will load the first `Size` ∗ `Unit` bits from the position *BinBase* + *BinOffset* and turn these bits into a float. The Flags argument specifies whether the bytes will be taken in big- or little-endian order.

Fail Conditions

- (*BinOffset* + `Size` ∗ `Unit`) > *BinSize*
- Size ∗ Unit is not equal to 32 or 64
- Wrong argument type

binary bs_get_binary(uint Size, uint Flags, uint Unit)

This BEAM instruction is used for the translation of a `Bin:Size/binary-`*specifier*`-unit:Unit` expression. It will create a *sub-binary* of size Size ∗ Unit if *BinOffset* is divisible by 8 (the creation of a sub-binary requires *BinOrig*; that is why that variable is a part of the state). Otherwise, it will create a new *heap* or *reference counted binary* and copy the first (Size ∗ Unit)/8 bytes into it. `Size` can also be the atom `all` in which case Size ∗ Unit will be interpreted as *BinSize* − *BinOffset*. Flags does not change the result of the operation but it contains the byte boundary information.

Fail Conditions

- (*BinOffset* + Size ∗ Unit) > *BinSize*
- Size = `all` and *BinOffset* is not evenly divisible by 8
- Size ∗ Unit is not evenly divisible by 8
- Wrong argument type

bs_save(StateLabel)

Saves the current state giving it the label StateLabel. This instruction cannot fail.

bs_restore(StateLabel)

Reloads a former state that has the label StateLabel. This instruction cannot fail.

bs_test_tail(NumberOfBits)

This instruction increases *BinOffset* by NumberOfBits and then checks if the resulting *BinOffset* is equal to *BinSize*. If the condition is met, the match is over.

Fail Conditions

- (*BinOffset* + NumberOfBits) ≠ *BinSize*

Example 3.1 The simple matching expression

```
<<X:16/integer-signed,
  Y:8/float-little-unit:8, Z/binary>> = Binary
```

gets compiled to the following pseudo-BEAM code where the names of the variables have not been changed from their ERLANG names. Also, FL shows the label that the flow will continue at if some instruction fails.

```
L0:  bs_start_match(Binary)          % FL = L1
     X = bs_get_integer(16, 5, 1)    % FL = L1
     Y = bs_get_float(8, 3, 8)       % FL = L1
     Z = bs_get_binary(all, 1, _)    % FL = L1
     bs_test_tail(0)                 % FL = L1
     %% here the binary match has succeeded
     ...
L1:  %% code to handle binary match failure
```

Let's look at the instructions' operands. Recall that in bs_get_* instructions the first operand is the Size, the second is the Flags, and the third denotes the Unit. For bs_get_integer and bs_get_float only the Flags operand needs to be explained. Both instructions are guaranteed to start at a byte boundary. The first instruction starts at the beginning of the binary, and the other starts after 16 bits (e.g. 2 bytes) into the binary. This means that the low bit will be set in each of their Flags operands. Also, the `signed` type specifier sets the third Flags bit. Thus, the Flags operand is 5 for bs_get_integer. Similarly, the `little` type specifier sets the second bit, which explains why the Flags operand of bs_get_float is 3. It is also clear that bs_get_binary is byte-aligned since it starts after $16+8*8 = 80$ bits (e.g. 10 bytes) and since endianism and signedness do not apply to binaries, the Flags operand is 1 in this case. Since a size is not specified for the third segment of the binary, the Size defaults to `all`. This also means that the Unit operand is irrelevant for bs_get_binary in this case (we denote this with an underscore).

Example 3.2 The translation of the ERLANG code of Example 2.2 is shown in Figure 1. The first branch of the `case` statement is translated to the instructions between labels L_0 and L_2. The first segment of the binary consists of the bs_get_integer and is_eq_exact instructions, and the second segment is translated to a bs_get_binary instruction. What is interesting to note is that if the bs_get_integer instruction fails, the execution will continue at the third branch (label L_3), not at the second. This happens because the fail conditions for a bs_get_integer instruction are exactly the same as those for a bs_skip_bits instruction. Because of this, the second branch can start with the state of the match after the bs_get_integer instruction. This is shown by the bs_save and bs_restore instructions.

4. TRANSLATION TO NATIVE CODE

The next step in the scheme is to translate the virtual machine instructions into native code. We do this by first translating the VM instructions to intermediate code (Icode). Then we proceed to translate the Icode into RTL code.

Many of the instructions can be quite tedious to translate for all possible argument value combinations. Therefore, a design decision was to translate to native code only commonly occurring argument combinations of VM instructions and to fall back to calling the interpreter functions in all other cases. This translation scheme requires that the state of the match is the same in the interpreter as in the native code environment. This is ensured by saving the state every time a call is made to the interpreter and to reload the state after the call has returned.

```
L0:  bs_start_match(Binary)          % FL = L5
     bs_save(1)
     x1 = bs_get_integer(8, 1, 1)    % FL = L3
     bs_save(0)
     is_eq_exact(x1, 13)             % FL = L2
     bs_restore(0)
     X = bs_get_binary(all, 1, _)    % FL = L2
     bs_test_tail(0)                 % FL = L2
     x0 = X
     jump L4
L2:  bs_restore(0)
     X = bs_get_integer(16, 1, 1)    % FL = L3
     bs_skip_bits(all, 1)            % FL = L3
     bs_test_tail(0)                 % FL = L3
     x0 = X
     jump L4
L3:  bs_restore(1)
     X = bs_get_binary(all, 1, _)    % FL = L5
     x0 = 'error'
L4:  %% here the binary match has succeeded
     ...
     return
L5:  %% code to handle binary match failure
```

Figure 1: Translation of binary pattern matching.

4.1 Translation to Icode

Except for some differences between the representations used at the Icode and at the virtual machine level, the translation from BEAM to Icode is quite straightforward. Two new instructions, bs_get_binary_all and bs_skip_bits_all, are added. They are used when the size argument is the atom `all`. The Unit argument is multiplied by the Size argument if Size is an integer. The result is then coded into the name of the operation. There are also five new argument registers added. Four of these represent the state of the matching and the fifth contains the address to the state structure in the interpreter. So for an arbitrary BEAM instruction the translation to Icode is as shown in Figure 2.

Three of these instructions (bs_skip_bits, bs_skip_bits_all, and bs_test_tail) do not influence all state variables, but only *BinSize* and *BinOffset* and thus the corresponding Icode instructions only get these two state variables as arguments. To simplify the notation we will refer to the state as an object called *State* and to its *BinOffset* variable as *State.Offset*. Similarly for other state variables.

A new memory management operation is also introduced in Icode. It is placed just before the bs_start_match instruction. This operation is discussed in Section 5.

4.2 Translation to RTL

We divide the Icode instructions into two groups: The first group contains bs_get_* instructions. The other group thus contains Icode instructions that do not read anything from the binary but only manipulate the control flow and the state. That is, it contains the bs_start_match, bs_skip_bits, bs_skip_bits_all, bs_save, bs_restore, and bs_test_tail instruc-

```
if Size is an integer ->
      {bs_operation, Size*Unit, Flags} (State_Address, BinOffset, BinSize, BinBase, BinOrig)
if Size is the atom all ->
      {bs_operation_all, Flags} (State_Address, BinOffset, BinSize, BinBase, BinOrig)
if Size is a variable ->
      {bs_operation, Unit, Flags} (Size, State_Address, BinOffset, BinSize, BinBase, BinOrig)
```

Figure 2: Translation of BEAM instruction bs_instruction(Size, Unit, Flags) **to Icode.**

tions. We will call the members of this group *non-reading Icode instructions*. Consequently we will call the members of the first group *reading Icode instructions*.

4.2.1 Utilities

We have written some utility functions that produce code for some common situations. They will be used in what we call pseudo-RTL instead of the code that they are creating.

Gen_make_size(FinalSize, Unit, SizeVar, FailLabel)
produces RTL code to multiply **Unit** with **SizeVar** and place the result in **FinalSize**. Since **Unit** is typically 1 or 8, multiplication is either trivial or can be strength reduced to a shift. If there is any problem with the content of **SizeVar**, execution will continue at **FailLabel**.

Gen_bs_call(Instruction)
produces RTL code to update the state of the interpreter, make a call to the function corresponding to **Instruction**, and to finally update the state registers when the call has returned.

Check_size(BinSize, Offset, Size, NewOffset, FailLabel)
produces RTL code to add **FinalSize** to **BinOffset** and put the result in **NewOffset**. If **NewOffset** is greater than **BinSize** the execution continues at **FailLabel**.

Load_bytes(Dst, Base, Offset, Signedness, Endianism, NoOfBytes)
produces RTL code that loads **NoOfBits** consecutive bytes from **Base+Offset** and adds **NoOfBytes-1** to **Offset**. The bytes are put in little- or big-endian order into the register **Dst** depending on the value of **Endianism**. **Signedness** specifies whether the loaded number should be interpreted as signed or unsigned. **NoOfBytes** is an integer between 1 and 4.

Load_state(Address, Base, Size, Offset, Orig)
produces RTL code to reload the values of the state registers from the state in the interpreter.

Make_sub_binary(Dst, Size, Offset, Orig)
produces RTL code to make a sub-binary on the heap. A tag is put in the first position, **Size** that is the size in bytes of the sub-binary is put in the second position, **Offset** which says how many bytes into the original binary the sub-binary starts is placed in the third position and **Orig** which is a pointer to the original binary is placed in the fourth and last position. Finally a tagged pointer to the first position is placed in **Dst**.

Make_float(Dst, HighBits, LowBits)
produces RTL code to make a float on the heap. A tag is put in the first position, the **HighBits** is put in the second position and the **LowBits** is placed in the third and last position. Then a tagged pointer to the first position is placed in **Dst**.

4.2.2 Translation of non-reading Icode instructions

In any binary matching operation, bs_start_match is the first instruction. It is translated to RTL code that first makes a call to the interpreter function to initialize the state. If this call fails, execution continues at FailLabel. If the call succeeds, code created by **Load_state** is executed to load the state from the interpreter into RTL registers r_0, ..., r_4 as will be shown in the example below.

The {bs_skip_bits, Bits}(*Bitvar*, *State*) instruction is always translated to RTL code that does not contain calls. The translation depends on if there is a *Bitvar* or not. If there is a *Bitvar*, the utility **Gen_make_size** is invoked to produce code that multiplies Bits with *Bitvar* and places the result in a temporary. If there is no *Bitvar* variable, then Bits is placed directly in the temporary. The utility **Check_size** generates code to test if the value of the temporary is smaller than or equal to the difference between *State.Size* and *State.Offset*. If that is the case, the temporary is added to *State.Offset* and the execution continues at SuccessLabel belonging to the operation. If that is not the case, execution will continue at the FailLabel belonging to the operation.

The {bs_skip_bits_all, Flags}(*State*) instruction is even simpler to translate since there is never any variable. If the first bit of Flags is set then the operation can not fail and *State.Offset* is just set to *State.Size*. If this is not the case a runtime check is performed to see if *State.Offset's* three low bits are equal to zero if this is the case again *State.Offset* is set to *State.Size* and execution continues at SuccessLabel. Otherwise execution continues at FailLabel.

The bs_save(*StateLabel*, *State*) instruction stores the current value of *State.Offset* at the address where *BinOffset* is stored in the interpreter. Then there is call to the corresponding function of the interpreter and finally the state is reloaded by code created by the **Load_state** utility.

The bs_restore(*StateLabel*, *State*) instruction makes a call to the function of the interpreter and reloads the state with code created by the **Load_state** utility.

Finally, the {bs_test_tail, Bits}(*State*) instruction adds Bits to *State.Offset* and puts the result into *State.Offset*. If *State.Size* = *State.Offset* execution continues at SuccessLabel; otherwise it continues at FailLabel.

Examples. In Table 2 we show the resulting pseudo-RTL code for some of the above Icode instructions. The representation is called pseudo-RTL since the code produced by the utility functions is only represented by the name of that function. The state variables are put into five RTL registers (r_0, ..., r_4) as follows:

$r_0 \leftarrow$ State.Address $r_3 \leftarrow$ State.Offset
$r_1 \leftarrow$ State.Base $r_4 \leftarrow$ State.Orig
$r_2 \leftarrow$ State.Size

Table 2: Translation of non-reading instructions to RTL.

(State) = bs_start_match(v_0)	{bs_skip_bits_all, 0}(State)	bs_save(1, State)
$r_5 \leftarrow$ **bs_start_match**(v_0) **[c] then** L_1 L_1: **if** (r_5 **eq 0) then L2 else FL** L_2: **Load_state**(r_0, r_1, r_2, r_3, r_4) **then SL**	$r_5 \leftarrow r_3$ **'and' 7 then** L_1 L_1: **if** (r_5 **eq 0) then** L_2 **else FL** L_2: $r_3 \leftarrow r_2$ **then SL**	[r_0 + **OFFSET**] $\leftarrow r_3$ **then** L_1 L_1: **bs_save(1) [c] then** L_2 L_2: **Load_state**(r_0, r_1, r_2, r_3, r_4) **then SL**
{bs_skip_bits, 8}(v_0, State)	{bs_test_tail, 8}(State)	bs_restore(1, State)
Gen_make_size(r_5, 8, v_0) **then** L_1 **else FL** L_1: **Check_size**(r_2, r_3, r_5, r_6) **then** L_2 **else FL** L_2: $r_3 \leftarrow r_6$ **then SL**	$r_5 \leftarrow r_3$ **add 8 then L1** L_1: **if** (r_5 **eq** r_2) **then SL else FL**	L_1: **bs_restore(1) [c] then** L_2 L_2: **Load_state**(r_0, r_1, r_2, r_3, r_4) **then SL**

The SuccessLabels and FailLabels associated with the different Icode operations will not be assigned proper label numbers but they will simply be called **SL** and **FL**.

4.2.3 Translation of reading Icode instructions

The **bs_get_binary_all** is the only instruction that is translated to native code for all argument combinations. On the other hand, for example the **bs_get_binary** instruction is only executed in native code if the result ends up being a sub-binary. The **bs_get_float** instruction is only executed in native code if it has an effective size of 64 and the **bs_get_integer** instruction is only executed in native code if it has an effective size that is less than 28 and it is big-endian. For several of the cases, this decision has to be made at runtime.

The translation of the {**bs_get_binary_all**, **Flags**}(*State*) instruction depends on the low bit of the **Flags** argument. When it is not set, a runtime test will check if the three low bits of *State.Offset* are equal to zero. If not, execution will continue at the FailLabel. Otherwise, both cases will continue in the same way by subtracting *State.Offset* from *State.Size* and place that result in an RTL temporary (**Tmp**$_1$) which is then right-shifted by 3 to get the size of the new binary in bytes. The offset in bytes of the new sub-binary is obtained in a similar way from *State.Offset* and is placed in another temporary (**Tmp**$_2$). After this, *State.Size* is moved to *State.Offset*. Finally, the **Make_binary** utility is used with the return variable, **Tmp**$_1$, **Tmp**$_2$, and *State.Orig* as arguments; see also Table 3.

The {**bs_get_binary**, **Size**, **Flags**}(*Sizevar*, *State*) instruction is translated to native code when the result is a sub-binary. Depending on whether there is a *Sizevar* or not, the translations are slightly different. If there is a *Sizevar*, the utility **Gen_make_size** is invoked to produce code that multiplies **Size** with *Sizevar* and place the result in a temporary (**Tmp**$_1$). If there is no *Sizevar*, **Size** is placed directly in **Tmp**$_1$. A check is made to see if the value in **Tmp**$_1$ is evenly divisible by eight. If this is not the case, execution continues at FailLabel. In case the low bit of **Flags** is not set, a runtime check is performed to see if *State.Offset* is evenly divisible by eight. If this is not the case, **Gen_bs_call** is invoked to call the interpreter, but if it is evenly divisible by eight the result of the match will be a sub-binary. However, we first have to check if there are enough bits left to perform the match. This is done by he **Check_size** utility, *State.Size* is used as the binsize argument, *State.Offset* as the offset argument, **Tmp**$_1$ as the size argument and a new temporary (**Tmp**$_2$) as the **NewOffset** argument. After this, **Tmp**$_1$ and *State.Offset* are right-shifted to get the size and offset of the sub-binary in bytes. Then the **Make_binary** utility is used with the return variable, **Tmp**$_1$, *State.Offset* and *State.Orig* as arguments. Finally **Tmp**$_2$ is moved into *State.Offset*.

The {**bs_get_float**, **Size**, **Flags**}(*Sizevar*, *State*) instruction is translated to native code only if the effective size is 64 and the binary is byte-aligned; otherwise a call to the interpreter function is made using **Gen_bs_call**. The effective size is constructed as in the **bs_get_binary** case and the result is placed in a temporary **Tmp**$_1$. (In case there is no *Sizevar*, the check that **Size** is 64 is done at compile time.) If the low bit in **Flags** is not set, a check is made to ensure that *State.Offset* is evenly divisible by eight. If these conditions are met, the **Check_size** utility is used with *State.Size*, *State.Offset*, and **Tmp**$_1$ as the first three arguments and a new temporary (**Tmp**$_2$) as the **NewOffset** argument. After this, *State.Offset* is right-shifted by 3 and **Load_bytes** is called with **Tmp**$_1$ as **Dst**, *State.Base* as **Base**, *State.Offset* as **Offset**, **Signedness** set to 'unsigned' and **Endianism** set to little or big as dictated by the second bit in **Flags**. **NoOfBytes** is set to four. Then *State.Offset* is increased by one and the **Load_bytes** utility is invoked again with the same arguments except that a new temporary (**Tmp**$_3$) is used instead of **Tmp**$_1$. Finally, the utility **Make_float** is called with the return variable, **Tmp**$_1$ and **Tmp**$_3$ as arguments, or in case it is little-endian with the reverse order of **Tmp**$_1$ and **Tmp**$_3$.

The {**bs_get_int**, **Size**, **Flags**}(*Sizevar*, *State*) instruction is the most complicated one to translate. To simplify the presentation, four different cases will be considered. When we write right*-shift below it means that it is an arithmetic right-shift if the signed flag is set and a logic right-shift if it is not set. The requirement that **Size** has to be smaller than 28 might seem extremely arbitrary, but it guarantees that the result can be stored as a tagged integer in one 32-bit machine word.

1. This case occurs when
   - **Size** < 28
   - No *Sizevar* argument exists.
   - Low bit set in **Flags**. (Byte alignment)

   Its translation is to use the **Check_size** utility with **Size**, *State.Size*, *State.Offset*, and a new temporary (**Tmp**$_1$) as arguments to see if there are enough bits left in the binary. *State.Offset* is right-shifted three positions. Then **Load_bytes** is used with a new temporary (**Tmp**$_2$) as **Dst**, *State.Base* as **Base**, *State.Offset* as **Offset**, **Signedness** set as dictated by the third bit in **Flags**, **Endianism** as dictated by the second bit in **Flags**, and **NoOfBits** as $\lceil$**Size**$/8\rceil$. **Tmp**$_1$ is then moved to *State.Offset*. To remove extra bits, **Tmp**$_2$ is right*-shifted (8 − **Size**) mod 8 positions. Finally, **Tmp**$_2$ is tagged[1] and the result is moved into the return register.

[1]The current tagging is left-shift four positions and an *or* with 15.

2. This case occurs when
   - Size < 28
   - No *Sizevar* argument exists.
   - Low bit not set in Flags. (No byte alignment)
   - Third bit not set in Flags. (big-endian)

   First we calculate the minimal number of bits that need to be loaded (found by masking the last three bits of *State.Offset* and putting the result into Tmp_1). Then Size is added and the result is placed in Tmp_2. If this result is larger than 32, a call is made to the interpreter using Gen_bs_call. After this, Check_size checks whether the instruction is OK. The same arguments are used as in case 1 except that the NewOffset argument is a new temporary (Tmp_3). A runtime check is performed to decide how many bytes should be loaded. The choice is restricted by Size to two different possibilities. The result is that $\lceil \mathsf{Tmp}_2/8 \rceil$ bytes are loaded. *State.Offset* is then right-shifted by 3 and Load_bytes is used to load the number of bytes that has been decided earlier. These bytes are loaded into Tmp_2. After that, Tmp_3 is moved into *State.Offset*. Then Tmp_2 is left-shifted $8*(4-$the number of bytes loaded). Then Tmp_2 is left-shifted again this time Tmp_1 positions. Finally Tmp_2 is right*-shifted $(28-\mathsf{Size})$ positions and the result of (Tmp_2 or 15) is placed in the return variable.

3. This case occurs when
   - A *Sizevar* argument is present.
   - Low bit set in Flags. (Byte alignment)
   - Third bit not set in Flags. (big-endian)

   The effective size is produced by Gen_make_size and put into Tmp_1. Tmp_1 is then checked to see if it is larger than 27. If that is the case a call is made to the emulator. Otherwise a number of bytes that is equal to the value of Tmp_1 right shifted 3 positions are loaded into Tmp_2. State.Offset is than increased by Tmp_1. After that Tmp_2 is right shifted (8 - Tmp_1 & 7) positions. Finally the value in Tmp_2 is tagged and moved into the return variable

4. This is the default case and is translated simply by a call to the corresponding interpreter function using Gen_bs_call.

Examples. Eight different examples of translation of reading instructions to pseudo-RTL code are shown in Table 3. One corresponding to the case when a bs_get_binary_all instruction is not guaranteed to be byte-aligned. Two different examples for the bs_get_binary instruction with and without a variable size argument. There are also two examples of the bs_get_float instruction. One that is big-endian and has a variable size argument and one with static size that is little-endian. There are three different examples of the bs_get_integer instruction; one for each of cases 1, 2, and 3.

The structure of the pseudo-RTL code is the same as before except that there is also a mapping of the different temporaries, the argument variables, and the return value to specific RTL registers as shown below:

$\mathsf{r}_5 \leftarrow \mathsf{Tmp}_1$ $\quad \mathsf{v}_0 \leftarrow$ Return value
$\mathsf{r}_6 \leftarrow \mathsf{Tmp}_2$ $\quad \mathsf{v}_1 \leftarrow$ Argument variable
$\mathsf{r}_7 \leftarrow \mathsf{Tmp}_3$

5. MEMORY MANAGEMENT

Several of the translated instructions might need space on the heap and can therefore trigger garbage collection. However, performing garbage collection in the middle of a binary matching operation is problematic because the matching state can contain pointers to the heap and these pointers are currently not considered as roots by the BEAM garbage collector. Modifying the code of the collector is of course an option, but since we want to minimize the changes to parts of the runtime system that are outside the control of HiPE, the only real option we have is to ensure that there is enough room on the heap when a matching begins.

Consider the beginning and the end of a binary matching operation as discussed in Section 3.1. With this definition we can find the graphs that describe the binary matches from a control flow graph of an entire function. These graphs have some special properties. They are acyclic, they have one start node and only the last instruction in each basic block can write to the heap. The last property follows from the fact that all reading instructions have Success- and Fail-Labels. Three of the instructions that can write to the heap have well-defined maximum space needs. For example, the maximum need for bs_get_float is 3 words. It is 4 for bs_get_binary_all and (MAX_HEAP_BIN_SIZE/4)+2 words for bs_get_binary. One can further decrease these maximal limits by looking at the Size and Flags arguments and whether or not there is a runtime argument. The situation is slightly more complicated for bs_get_int, because this instruction may need an arbitrary amount of heap space since an ERLANG bignum can contain an arbitrary number of words. It seems inevitable that the maximum heap need calculation needs to be done at runtime in this case.

It is however undesirable to have to do complicated calculations every time a binary matching starts. Hence, to avoid this we separate the heap need for instructions with well defined maximal heap needs (*heapneed1*) from the heap need of other instructions (*heapneed2*). The calculation of *heapneed1* is simple and inductively defined: By assigning a heap need in words to each node in the match graph, the maximum heap need of a leaf node is equal to the heap need of that node and then the maximum heap need of a non-leaf node can be calculated as the heap need of a node plus the maximum heap need of the nodes' successors. The maximal heap need of a match is thus given by the maximum heap need of the node containing the bs_start_match instruction.

To get a conservative estimate of *heapneed2* (i.e., an estimate guaranteed to be larger than or equal to the actual value of *heapneed2*) we just sum the individual heap needs of each of the instructions that contribute to *heapneed2*. That this is a conservative estimate follows from the fact that the match graph is acyclic and therefore each instruction can only be executed once. To perform this addition at runtime is a simple feat and when this has been done *heapneed1* and *heapneed2* are added and a check is performed to see whether there is enough room on the heap to perform the match or whether a call to the garbage collector has to precede the match. (Before the matching begins, a garbage collection is possible.)

6. PERFORMANCE

The performance evaluation of our implementation is divided into three parts: The first part tests the speed of exe-

Table 3: Translation of reading instructions to RTL.

v_0 ={bs_get_binary_all, 0}(State)	v_0 ={bs_get_binary, 40, 1}(State)
$r_5 \leftarrow (r_3$ 'and' 7) if (r_5 eq 0) then L_1 else FL L_1: $r_5 \leftarrow (r_2$ sub $r_3)$ $r_5 \leftarrow (r_5$ srl 3) $r_6 \leftarrow (r_3$ srl 3) $r_3 \leftarrow r_2$ Make_sub_binary(v_0, r_5, r_6, r_4) then SL	$r_5 \leftarrow 40$ $r_6 \leftarrow (r_5$ 'and' 7) if (r_6 eq 0) then L_1 else FL L_1: Check_size(r_2, r_3, r_5, r_6) then L_2 else FL L_2: $r_5 \leftarrow (r_5$ srl 3) $r_3 \leftarrow (r_3$ srl 3) Make_sub_binary(v_0, r_5, r_3, r_4) $r_3 \leftarrow r_6$ then SL
v_0 ={bs_get_binary, 8, 0}(v_1, State)	v_0 ={bs_get_float, 8, 0}(v_1, State)

Gen_make_size(r_5, 8, v_1) then L_1 else FL
L_1: $r_6 \leftarrow (r_5$ 'and' 7)
if (r_6 eq 0) then L_2 else FL
L_2: $r_6 \leftarrow (r_3$ 'and' 7)
if (r_6 eq 0) then L_3 else L_5
L_3: Check_size(r_2, r_3, r_5, r_6) then L_4 else FL
L_4: $r_5 \leftarrow (r_5$ srl 3)
$r_3 \leftarrow (r_3$ srl 3)
Make_sub_binary(v_0, r_5, r_3, r_4)
$r_3 \leftarrow r_6$ then SL
L_5: Gen_bs_call({bs_get_binary,8,0}) then SL else FL
	Gen_make_size(r_5, 8, v_1) then L_1 else FL
	L_1: if (r_5 eq 64) then L_2 else L_5
	L_2: $r_6 \leftarrow (r_3$ 'and' 7)
	if (r_6 eq 0) then L_3 else L_5
	L_3: Check_size(r_2, r_3, r_5, r_6) then L_4 else FL
	L_4: $r_3 \leftarrow (r_3$ srl 3)
	Load_bytes(r_5, r_1, r_3, unsigned, big, 4)
	$r_3 \leftarrow (r_3$ add 1)
	Load_bytes(r_7, r_1, r_3, unsigned, big, 4)
	Make_float(v_0, r_5, r_7) then SL
	L_5: Gen_bs_call({bs_get_float,8,0}) then SL else FL

v_0 ={bs_get_float, 64, 3}(State)	v_0 ={bs_get_integer, 10, 5}(State)
$r_5 \leftarrow 64$ Check_size(r_2, r_3, r_5, r_6) then L_1 else FL L_1: $r_3 \leftarrow (r_3$ srl 3) Load_bytes(r_5, r_1, r_3, unsigned, little, 4) $r_3 \leftarrow (r_3$ add 1) Load_bytes(r_7, r_1, r_3, unsigned, little, 4) Make_float(v_0, r_7, r_5) then SL	Check_size(r_2, r_3, 10, r_5) then L_1 else FL L_1: $r_3 \leftarrow (r_3$ srl 3) Load_bytes(r_6, r_1, r_3, signed, big, 2) $r_3 \leftarrow r_5$ $r_6 \leftarrow (r_6$ sra 6) $r_6 \leftarrow (r_6$ sll 4) $v_0 \leftarrow (r_6$ 'or' 15) then SL
v_0 ={bs_get_integer, 20, 0}(State)	v_0 ={bs_get_integer, 1, 1}(v_1, State)

$r_5 \leftarrow (r_3$ 'and' 7)
$r_6 \leftarrow (r_5$ add 20)
if (r_6 le 32) then L_1 else L_6
L_1: Check_size(r_2, r_3, 17, r_7) then L_2 else FL
L_2: $r_3 \leftarrow (r_3$ srl 3)
if (r_6 le 24) then L_3 else L_4
L_3: Load_bytes(r_6, r_1, r_3, unsigned, big, 3)
$r_6 \leftarrow (r_6$ sll 8) then L_5
L_4: Load_bytes(r_6, r_1, r_3, unsigned, big, 4) then L_5
L_5: $r_3 \leftarrow r_7$
$r_6 \leftarrow (r_6$ sll $r_5)$
$r_6 \leftarrow (r_6$ srl 8)
$v_0 \leftarrow (r_6$ 'or' 15) then SL
L_6: Gen_bs_call({bs_get_int,20,0}) then SL else FL
	Gen_make_size(r_7, 1, v_1) then L_1 else FL
	L_1: if (r_7 lt 28) then L_2 else L_{11}
	L_2: Check_size(r_2, r_3, r_7, r_5) then L_3 else FL
	L_3: $r_3 \leftarrow (r_3$ srl 3)
	if (r_7 le 8) then L_4 else L_5
	L_4: Load_bytes(r_6, r_1, r_3, unsigned, big, 1) then L_{10}
	L_5: if (r_7 le 16) then L_6 else L_7
	L_6: Load_bytes(r_6, r_1, r_3, unsigned, big, 2) then L_{10}
	L_7: if (r_7 le 24) then L_8 else L_9
	L_8: Load_bytes(r_6, r_1, r_3, unsigned, big, 3) then L_{10}
	L_9: Load_bytes(r_6, r_1, r_3, unsigned, big, 4) then L_{10}
	L_{10}: $r_3 \leftarrow r_5$
	$r_7 \leftarrow (r_7$ and 7)
	$r_7 \leftarrow (8$ sub $r_7)$
	$r_6 \leftarrow (r_6$ srl $r_7)$
	$r_6 \leftarrow (r_6$ sll 4)
	$v_0 \leftarrow (r_6$ 'or' 15) then SL
	L_{11}: Gen_bs_call({bs_get_int,1,1}) then SL else FL

cution of two telecom applications supplied to us by Ericsson depending on the compilation scheme that is used. The second part quantifies how alignment influences the speed of execution for different translation schemes. The program used is a small synthetic benchmark that parses a binary into integers, floats and binaries; to do so, it uses all the different ***BEAM*** instructions that handle binaries. The third part examines how the speed of execution of an ERLANG program that does not use the bit syntax compares to one that does. All performance tests have been conducted on a machine with a 266 MHz Pentium processor running Linux.

6.1 Application performance

We evaluated the performance of code compiled with the scheme presented in this paper (***HiPE v 1.2***), comparing it to that of the ***BEAM*** and of ***HiPE v 1.1*** (the version of HiPE included in Erlang/OTP R8B) that does not compile bit syntax operations to native code but instead treats them as built-in functions.

The benchmarks used are functions from the core of two protocol handling ERLANG applications supplied to us by Ericsson. The extract_gtp_c_message (*extract*) function takes

Table 4: Time performance for the extract and decode benchmarks (in seconds).

	BEAM	***HiPE v 1.1***	***HiPE v 1.2***
extract	10.3	9.63	4.87
decode	142.2	191.3	141.2

Table 5: Time performance for the synthetic benchmark (in seconds).

	BEAM	***HiPE v 1.2***
aligned, static size	37.3	16.7
unaligned, static size	58.3	38.8
aligned, variable size	58.4	30.2
unaligned, variable size	58.0	52.1

a binary containing a GTP_C message as input, extracts the information from the message header, and returns it. The decode_gtp_c_message (*decode*) function also takes a binary containing a GTP_C message as input, but this function translates the entire message into a record.

Execution times for the functions with the different compilation schemes are shown in Table 4. It is clear that the new compilation scheme outperforms the old one for both functions. For the *extract*, native code compilation is significantly faster than calling C functions (***BEAM***) as well. This is probably because most of the instructions in the *extract* program are quite simple, so a lot of calculations can be inlined to native code which favors this approach.

In *decode*, the difference between ***HiPE v 1.2*** and ***BEAM*** is less than 1%. The reason for this small difference is probably because this program contains quite complex binary instructions. Some of these are not translated to native code at all, and the remaining ones are translated to complicated native code that quite possibly is not significantly more efficient than the corresponding machine code generated by the GNU C compiler.

6.2 Impact of byte alignment

We have devised a small synthetic benchmark to display the properties of the new compilation scheme with respect to byte alignment. The program takes a binary and parses it into three integers, two binaries and a float. One of the integers is used to decide the flow of execution. The benchmark also uses bs_skip_bits instructions. The program exists in four different versions. One where all operations are byte-aligned, one where only the last operation is byte-aligned, one where the sizes are given as arguments but the `unit` specifier is used to guarantee byte-alignment, and finally one where the sizes are given as arguments but the `unit` specifiers do not guarantee alignment.

We have run this benchmark compiled with ***HiPE v 1.2*** and with the ***BEAM***. The results, presented in Table 5, show that the scheme presented in this paper is quite sensitive to the alignment of the operations. In contrast, the ***BEAM*** has about the same execution speed for three of the versions of the program and is only slightly faster for the first version. In ***HiPE v 1.2*** the performance of the unaligned versions are significantly lower than that of the aligned versions. The conclusion is that the new scheme favors code that respects byte alignment, but it is faster than the ***BEAM*** even if the code is not structured in this way.

Table 6: Time performance for the broadband benchmark (in seconds).

	BEAM	***HiPE v 1.2***
broadband-Bif	24.5	16.5
broadband-bs	16.2	6.77

6.3 Bit syntax versus Bifs

The last part of the evaluation compares the performance of an ERLANG program that uses the bit syntax with that of an equivalent program using built-in functions (Bifs) such as `split_binary` and `binary_to_list` to manipulate binaries. The program (originally using Bifs) was supplied to us by Ericsson and we have translated it to a program that uses the bit syntax. This program, just like the *decode* program in Section 6.1, takes a binary and parses it into a record. The program implements a broadband protocol. We refer to the two different versions of the program as *broadband-bs* and *broadband-Bif*.

The different versions were run with ***HiPE v 1.2*** and ***BEAM***. Timings are shown in Table 6. The results show a speedup of about a factor four when changing to using the bit syntax and compiling with ***HiPE v 1.2***. A reason for this is that the translation from the Bifs that manipulated binaries to the bit syntax has been quite straightforward. This automatically conserves byte boundary alignment since one could only operate on bytes with the old Bifs that manipulate binaries.

Acknowledgments

This research has been supported in part by the ASTEC (Advanced Software Technology) competence center with matching funds by Ericsson Development.

7. REFERENCES

[1] E. Johansson, M. Pettersson, and K. Sagonas. HiPE: A High Performance Erlang system. In *Proceedings of the ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM Press, Sept. 2000.

[2] P. Nyblom. The bit syntax - the released version. In *Proceedings of the Sixth International Erlang/OTP User Conference*, Oct. 2000. Available at http://www.erlang.se/euc/00/.

[3] M. Pettersson, K. Sagonas, and E. Johansson. The HiPE/x86 Erlang compiler: System description and performance evaluation. In *Proceedings of the Sixth International Symposium on Functional and Logic Programming*, LNCS. Springer, Sept. 2002.

[4] C. Wikström and T. Rogvall. Protocol programming in Erlang using binaries. In *Proceedings of the Fifth International Erlang/OTP User Conference*, Oct. 1999. Available at http://www.erlang.se/euc/99/.