Uppsala Master's Theses in
Computing Science 229
Examensarbete TF3
2002-10-22
ISSN 1100–1836

Compilation of Floating Point Arithmetic in the High Performance ERLANG Compiler.

Tobias Lindahl

Information Technology
Computing Science Department
Uppsala University
Box 311
S-751 05 Uppsala
Sweden

Supervisor and examiner: Konstantinos Sagonas

Passed:

Abstract. In the context of the dynamically typed concurrent functional programming language ERLANG, we describe a simple static analysis for identifying variables containing floating point numbers, how this information is used by the BEAM compiler, and a scheme for efficient (just-in-time) compilation of floating point bytecode instructions to native code. The attractiveness of the scheme lies in its implementation simplicity, it has been fully incorporated in Erlang/OTP R9, and improves the performance of ERLANG programs manipulating floats considerably. We also show that by using this scheme, Erlang/OTP, despite being an implementation of a dynamically typed language, achieves performance which is competitive with that of state-of-the-art implementations of strongly typed strict functional languages on floating point intensive programs.

1 Introduction

In dynamically typed languages the implementation of built-in arithmetic typically involves runtime type tests to ensure that the calculations which are performed are meaningful, i.e., that one does not succeed in dividing atoms by lists. Some of these tests are strictly necessary to ensure correctness, but the same variable can be repeatedly tested because the type information is typically lost after an operation has been performed. Removing these redundant tests improves execution time both by avoiding their runtime cost and by simplifying the task of the compiler (removing conditional branches simplifies the control flow graphs and allows the compiler to work with bigger basic blocks).

Of course, one way of attempting to solve this problem is to attack it at its root: impose a type system to the language and do (inter-modular) type inference. However, doing so is most often not trivial, might significantly slow down compilation, and hamper rapid prototyping. More importantly, type systems and powerful static analyses might not necessarily be in accordance with certain features deemed important for intended application domains (e.g., on-the-fly selective code updates that might invalidate the results of previous analyses), design decisions of the underlying implementation (e.g., the ability to selectively compile a *single* function at a time in a just-in-time fashion), or the overall philosophy of the language.

In this report, rather than addressing philosophical issues of programming language design, a more pragmatic approach to alleviating the downsides that absence of type information has for a native code compiler is taken. Specifically, a simple scheme for using *local* type analysis (i.e., the analysis is restricted to a single function) to identify variables containing floating point values is described. Moreover, this scheme has been fully incorporated in an industrial-strength implementation of a functional programming language (the Erlang/OTP system) and the performance gains that it offers is extensively quantified both in execution of virtual machine bytecode and of native code.

To make this report relatively self-contained, it starts with a brief presentation of ERLANG's characteristics (Section 2) followed by a brief description of the architecture of the HiPE just-in-time native code compiler (Section 3). In Section 4 a simple scheme to identify variables containing floating point values is presented, and the floating point aware translation of built-in arithmetic in the BEAM virtual machine instruction set is compared to its older translation. Section 5 contains a detailed account of how the HiPE compiler translates floating point instructions of the BEAM from its intermediate representation all the way down to both its SPARC and x86 back-ends, and how the features of the corresponding architectures are effectively utilized. The report ends with an evaluation of the performance of using the presented scheme both within different implementations of ERLANG and when compared with a state-of-the-art implementation of a strict statically typed functional language.

2 The Erlang Language and Erlang/OTP

ERLANG is a dynamically typed, strict, concurrent functional language. The basic data types include atoms, numbers, and process identifiers; compound data types are lists and

tuples. There are no assignments or mutable data structures. Functions are defined as sets of guarded clauses, and clause selection is done by pattern matching. Iterations are expressed as tail-recursive function calls, and ERLANG consequently requires tailcall optimization. ERLANG also has a catch/throw-style exception mechanism. ERLANG processes are created dynamically, and applications tend to use many of them. Processes communicate through asynchronous message passing: each process has a *mailbox* in which incoming messages are stored, and messages are retrieved from the mailbox by pattern matching. Messages can be arbitrary ERLANG values. ERLANG implementations must provide automatic memory management, and the soft real-time nature of the language calls for bounded-time garbage collection techniques.

Erlang/OTP is the standard implementation of the language. It combines ERLANG with the Open Telecom Platform (OTP) middleware, a library with standard components for telecommunications applications. Erlang/OTP is currently used industrially by Ericsson Telecom and other software and telecommunications companies around the world for the development of high-availability servers and networking equipment. Additional information about ERLANG can be found at `www.erlang.org`.

3 HiPE: Brief System Overview

HiPE is included as an optional extension in the open source Erlang/OTP system. It consists of a compiler from BEAM virtual machine bytecode to native machine code (currently UltraSPARC or x86), and extensions to the runtime system to support mixing interpreted and native code execution, at the granularity of individual functions.

BEAM. The BEAM intermediate representation is a symbolic version of the BEAM virtual machine bytecode, and is produced by disassembling the functions or module being compiled. BEAM is a register-based virtual machine which operates on a largely implicit heap and call-stack, a set of global registers for values that do not survive function calls (X-registers), and a set of slots in the current stack frame (Y-registers). BEAM is semi-functional: composite values are immutable, but registers and stack slots can be assigned freely.

BEAM to Icode. Icode is an idealized Erlang assembly language. The stack is implicit, any number of temporaries may be used, and all temporaries survive function calls. Most computations are expressed as function calls. All bookkeeping operations, including memory management and process scheduling, are implicit.

BEAM is translated to Icode mostly one instruction at a time. However, function calls and the creation of tuples are sequences of instructions in BEAM but single instructions in Icode, requiring the translator to recognize those sequences. The Icode form is then improved by application of constant propagation, constant folding, and dead-code elimination [5]. Temporaries are also renamed through conversion to a static single assignment form [1], to avoid false dependencies between different live ranges.

Icode to RTL. RTL is a generic three-address register transfer language. RTL itself is target-independent, but the code is target-specific, due to references to target-specific registers and primitive procedures. RTL has tagged registers for proper Erlang values, and untagged registers for arbitrary machine values. To simplify the garbage collector interface, function calls only preserve live tagged registers.

In the translation from Icode to RTL, many operations (e.g., arithmetic, data construction, or tests) are inlined. Data tagging operations are made explicit, data accesses and initializations are turned into loads and stores, etc. Optimizations applied to RTL include common subexpression elimination, constant propagation and folding, and merging of heap overflow tests.

The final step in the compilation is translation from RTL to native machine code of the target back-end (as mentioned, currently SPARC V8+ or IA-32).

4 Identification and Handling of Floats in the BEAM Interpreter

In this report, no formal definition of the static analysis that is used, but instead its basic ideas and how the analysis information is used in the BEAM interpreter are explained with the following example.

Example 1. Consider the ERLANG code shown in Fig. 1(a). Its translation to BEAM code without taking advantage of the fact that certain operands to arithmetic expressions are floating point numbers is shown in Fig. 1(b). Note that the code uses the general arithmetic instructions of the BEAM. These instructions have to test at runtime that their operands (constants and X-registers in this case) contain numbers, untag and possibly unbox these operands, perform the corresponding arithmetic operation, tag and possibly box the result storing it in the X-register shown on the left hand side of the arrow. Note that if such an arithmetic operation results in either a type error or an arithmetic exception, execution will continue at the fail label denoted by L_e.

```
-module(example).
-export([f/3]).

f(A,B,C) when is_float(C) ->
    X = A + 3.14,
    Y = B / 2,
    R = C * X - Y.
```

(a) ERLANG code.

```
is_float x2                          Lc
x0 ← arith '+' x0 {float,3.14}       Le
x1 ← arith '/' x1 {integer,2}        Le
x2 ← arith '*' x2 x0                 Le
x0 ← arith '-' x2 x1                 Le
return
```

(b) BEAM instructions for `f/3`.

Fig. 1. Naive translation of floating point arithmetic to BEAM bytecode.

Note however that even though ERLANG is a dynamically typed language, there is enough information in the above ERLANG code to deduce through a simple static analysis that certain arithmetic operations take floating point numbers as operands and return floating point numbers as results. For example, after the type test guard succeeds, it is known that variable `C` (argument register x_2) contains a floating point number. Because of the floating point constant 3.14, if the addition will not result in either a type error or an exception, it is clear that variable `X` will also be bound to a float. Similarly, because of the use of the floating point division operator, variable `Y` will also be bound to a float if successful, etc. Using the results of such an analysis could allow generation of the more efficient BEAM code shown in Fig. 2. Note that a new set of floating point registers (F-registers) has been introduced. These registers contain untagged floats.

```
is_float x2                    Lc
f0 ← fconv  x0
f1 ← fmove  {float,3.14}
fclearerror
f0 ← fadd   f0 f1              Le
f2 ← fconv  x1
f3 ← fconv  {integer,2}
f2 ← fdiv   f2 f3              Le
f4 ← fmove  x2
f4 ← fmul   f4 f0              Le
f0 ← fsub   f4 f2              Le
fcheckerror                    Le
x0 ← fmove f0
return
```

Fig. 2. Floating-point aware translation of `f/3` to BEAM bytecode.

As shown in this example, in recent versions of the BEAM, a separate set of instructions for handling floating point arithmetic has been introduced. Whenever it can be determined that the type of a variable is indeed a float, a block of floating point operations is created limited by fclearerror and fcheckerror instructions. Inside this block no type tests are needed for the variables marked as floats. The complete set of BEAM instructions for handling floats is shown in Table 1.

Table 1. BEAM floating point instructions.

Instruction	Description
fclearerror	Clear any earlier floating point exceptions.
fcheckerror	Check for floating point exception.
fconv	Convert a number to a floating point.
fadd	Floating point addition.
fsub	Floating point subtraction.
fdiv	Floating point division.
fmul	Floating point multiplication.
fnegate	Negate a floating point number.
fmove	Move between floating point registers and ordinary registers.

5 Handling of Floats in the HiPE Native Code Compiler

In the BEAM, whenever it is not known that a particular virtual machine register contains a floating point number, the float value is boxed, i.e., stored on the heap with a header word pointed to by the address kept in the register representing the number. Furthermore the address is tagged to show that the register is bound to a boxed value; see Fig. 3.

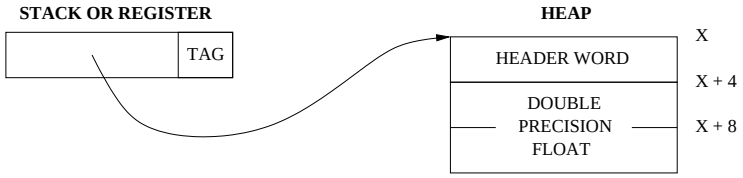

Fig. 3. A boxed float in the BEAM.

Whenever the float is used, the address has to be untagged, the header word has to be examined to find out the type of the variable (because e.g. tuples and bignums are boxed in the same manner), and finally the actual number can be used. Depending on the target architecture, the float is placed in the SPARC's floating point registers or on the x87 floating point stack, the computation takes place and then the result is boxed again and put on the heap. If the result is to be used again, which is typically the case, it has to be unboxed again prior to its use just as described above.

However, inside a basic block that is known to consist of floating point computations, all floating point numbers can be kept unboxed in the F-registers which are loaded either in the floating point unit (e.g., on the SPARC) or on the floating point stack of the machine (e.g., on the x86), thus removing the need of type testing each time the value is used. Furthermore, if a result of a computation is to be used again it can simply remain unboxed instead of being put on the heap and then read into the FPU again.

5.1 Translation to Icode

In the translation from BEAM bytecode to Icode most of the instructions are more or less kept unchanged and just passed on to RTL. The exception is fmove that either moves a value from an ordinary X-register to a floating point one (in which case it corresponds to an untagging operation), or vice versa (in which case it corresponds to a tagging operation). To handle the first case, Icode introduces the operation **unsafe_untag_float** and in the second **unsafe_tag_float**. These Icode operations will be expanded on the RTL-level as described below.

5.2 Translation to RTL

Translation of boxing and unboxing. When translating the **unsafe_untag_float** instruction, since it is known that the X-register contains a float, there is no need to examine the header word. The untagging operation can be performed by simply subtracting the float tag which currently is 2; see [6]. As can be seen in Fig. 3 the actual floating point value is stored at an offset of 4 from the untagged address, so instead of being translated to a subtraction of 2 and a **fload** with offset 4, **unsafe_untag_float** is translated to **fload** with offset 2, thus eliminating the actual untagging.

The **unsafe_tag_float** instruction writes the value to the heap, places a header word showing that this is a float, and finally tags the pointer with 2 to show that the value is boxed. Normally the garbage collection test that should be done to ensure that there is space on the heap is handled by a coalesced heap test, but otherwise one is added here.

Translation of floating point conversion. On converting an ERLANG number to its floating point representation it is essential to find out what the old representation was. The legal conversions are from integers, bignums, and possibly other floats. The reason the last case can occur is that the static analysis currently used does not discover all variables containing floats. These do not, of course, need to be converted but implicit in the fconv instruction is also the request to untag the value so this case is turned into an **unsafe_untag_float**.

The conversion from an integer is supported in both back-ends so this operation is kept as an fconv-instruction, but when the value is a bignum the operation is not inlined. Instead the instruction is turned into a call to the **conv_big_to_float** primary operation (primop) that returns a boxed float that needs to be untagged before further processing.

The separate handling of different types of conversion constitutes the only branches in the control flow graph (CFG) where there can be unboxed floats in registers. All functions can branch to a fail label but as discussed below all unboxed floats must be saved on the stack on function calls. Furthermore, if there is a comparison of floats the computational block is ended and the comparison is made on boxed values. Currently, there is no support for unboxed comparison.

Translation of error handling. The instructions fclearerror and fcheckerror are just setting and reading a variable in a C structure of the runtime system. Because HiPE currently does not support accessing information from C, they are implemented as calls to primops which is suboptimal in more than one aspect. Not only is a call to a primop not as cheap as reading the variable, but it also affects the spilling behavior. As mentioned all floats are spilled on the stack on function calls. A typical (extended) basic block of floating point computations has the following structure (in BEAM code):

```
f0 ← fmove x0
f1 ← fmove {float,3.14}
...
fclearerror
f2 ← fadd f0 f1          Le
...
fcheckerror              Le
x1 ← fmove f2
x1 ← fmove f3
...
```

The above will result in floats first being unboxed and put in floating point registers by the fmove instructions and then immediately being spilled since fclearerror is not inlined and requires a function call. The translation of the code around the fcheckerror instruction is similar. (In addition to the reading of the exception flag, fcheckerror also expands into the actual check whether there has been an exception or not.)

Translation of floating point arithmetic. fadd, fsub, fdiv, and fmul do not have to be treated in any special way. They are just propagated to the back-end. In the SPARC back-end the `fmov` SPARC instruction has a flag telling the processor if the value is to

be negated in addition to being moved. The `fnegate` instruction is therefore translated to a `fmov` which sets that flag.

5.3 Handling of floats in the SPARC back-end

Use of the SPARC floating point registers. The SPARC has 32 double precision floating point registers, half of which can instead be used as single precision registers in which case there are 32 single precision and 16 double precision floating point registers. On loading or storing double precision floats the address must be double word aligned, or the operation will result in a fault. Since there is no guarantee of such an alignment in neither BEAM nor HiPE, the fact that a double precision register is made up of two single precision ones is used and the instruction is turned into two single precision loads.

If the exclusive double precision registers need to be used, the only way to safely load to them would be to use two scratch single precision registers and then move the double precision value. This is not done, so these 16 registers are not being used.

The register allocation of the pseudo floating point variables to the real registers is handled by the linear scan register allocation algorithm in a slightly variated implementation (i.e., mapping to floating point spill slots with double word size) of the one which handles the general purpose registers in HiPE [7, 3].

Floating point numbers on the native stack. Floats are spilled to the stack when too many of them are live at the same time, but also whenever they are live over a function call. Since there are no guarantees that the called function does not use the floating point registers, their contents must be saved on the stack and then restored on return from the function. Currently, an extra pass through the CFG removes any redundant stores and loads.

On spilling floats to the native stack it must be ensured that the stack slots are marked as dead since the values are not tagged. If a float would be marked as live the garbage collector would try to follow the address it thought the stack slot pointed to and get a meaningless result.

There is one more case where untagged values are put on the stack. When converting a single word integer to a float the value typically resides in an ordinary register. SPARC handles the conversion by loading the integer value into a single precision floating point register and then converting it into the corresponding double precision register. However, the load instruction cannot use a register as source or it would interpret the value as an address, so the value is stored on the stack first.

Performing the operations. When a floating point operation is called all three of its operands must be in floating point registers. The SPARC, unlike the x86, has no support for letting one or more of the operands be a memory reference so two registers need to be available for the case when the two operands reside in memory.

A design decision of the HiPE compiler is to preserve the observable behavior of ERLANG programs. This includes preserving side-effects of arithmetic operations such as floating point exceptions; in ERLANG these can be caught by a `catch` statement.

Therefore, even in cases were the result of a floating point arithmetic operation is not needed, the operation can be eliminated only if it can be proved that it will not raise an exception. However, note that when a floating point operation is performed only for its side effects and its result is never used, the latter can safely be left in the register since SPARC does not demand the registers to be empty on leaving a function. If the result is to be used and the pseudo variable tied to the float is spilled, the result is stored in a stack slot. Currently, no test is made to see if the result is the operand of the next floating point operation that needs the scratch registers since this would require another pass through the code. (This would interfere with the JIT nature of the HiPE compiler.)

5.4 Handling of floats in the x86 back-end

Use of the x87 floating point unit. On the x86, all floating point operations are performed in the x87 floating point unit. The x87 is used as a stack with eight slots represented by `%st(i)`, $0 \leq$ `i` ≤ 7. In this section, whenever the stack is mentioned the x87 floating point stack is what is meant unless otherwise stated.

On the SPARC the pseudo variables can be globally mapped to floating point registers but because of the stack representation of the x87 the bindings between pseudo variables and stack slots are local to each program point.

Mapping to the x87. The approach of the mapping is based on the algorithm proposed in [4].

1. As in the SPARC back-end, using a variation of the linear scan register allocation algorithm, the floating point variables are mapped to seven pseudo stack slots. These do not represent the actual slots but this mapping is a way to ensure that at all times the unspilled values and a scratch value fit on the stack.
2. The CFG is traversed trace-wise: by starting from the beginning each successor is handled until the trace either merges with a trace already handled or reaches its end. In each basic block the instructions are transformed to operate on the actual stack positions and to add the pushing and popping behaviours. The mapping from pseudo variables to stack positions is propagated to the next basic block.
3. Whenever two traces are merged their mappings are compared. If they differ the adjoining trace is altered since the basic block and its successors already have been handled. This is done by adding a basic block containing stack shuffling code that synchronizes the mappings.
4. If a floating point instruction branches to a fail label the mapping that is kept at compile time may be corrupt since there is no way of knowing where the error occurred. The stack must then be completely freed so as to assure that it contains no garbage. This is done in the same basic block as the fail code since these operations are independent of the predecessor.

Translating the instructions. The top of the stack is represented by `%st(0)` and this slot is the only one that can interact with memory on loads and stores but also when using a memory cell as an operand. This can at times be inconvenient but an instruction

to switch places between the top and an arbitrary position `i` is available, `fxch %st(i)`. When used in conjunction with another floating point operation this instruction is very cheap. Only the source operand (src) of a floating point instruction can be a memory reference, so a spilled src is not pushed prior to its use. The destination operand (dst) must be on the stack so a spilled value can already be on the stack if it has been used as dst in an earlier instruction.

The liveness of each value is known at each point. A value that is not live out is immediately popped, but as described above a value that *is* live out is not necessarily pushed. A spilled value is not written back to its spill position unless it has to be popped. This means that there can be several spilled values on the stack at the same time. When a value is to be pushed and the stack is full a spilled value is popped and written back.

The instructions that work on the x87 allow pushing, popping, or simply working on the existing elements on the stack. Most instructions have both an ordinary and a popping version making it possible to both perform an operation and then pop one of the operands with one single instruction.

Example 2. Suppose the following calculation is to be performed.

$$X = A * B(A + C) + D$$

Using the pseudo variables $\texttt{\%f}_i$, $\texttt{i} \in \mathbf{N}$, the calculation corresponds to the following sequence of pseudo RTL instructions:

fmov A $\texttt{\%f}_0$
fmov B $\texttt{\%f}_1$
fmov C $\texttt{\%f}_2$
fmov D $\texttt{\%f}_3$
fadd $\texttt{\%f}_0$ $\texttt{\%f}_2$ $\texttt{\%f}_4$
fmul $\texttt{\%f}_0$ $\texttt{\%f}_1$ $\texttt{\%f}_5$
fmul $\texttt{\%f}_4$ $\texttt{\%f}_5$ $\texttt{\%f}_6$
fadd $\texttt{\%f}_6$ $\texttt{\%f}_3$ $\texttt{\%f}_7$
fmov $\texttt{\%f}_7$ **X**

After register allocation (where the index of $\texttt{\%f}_i$ has been limited to $0 \leq \texttt{i} \leq 7$) and translation to the two address code that the x86 uses, the above sequence becomes:

```
fmov A, %f0
fmov B, %f1
fmov C, %f2
fmov D, %f3
fadd %f0, %f2
fmul %f0, %f1
fmul %f1, %f2
fadd %f3, %f2
fmov %f2, X
```

Transforming this into real code for the x87:

Instruction	Stack
`fld A`	`[A]`
`fld B`	`[B,A]`
`fld C`	`[C,B,A]`
`fld D`	`[D,C,B,A]`
`fxch %st(3)`	`[A,C,B,D]`
`fadd %st(1), %st(0)`	`[A,A+C,B,D]`
`fmulp %st(2), %st(0)`	`[A+C,A*B,D]`
`fmulp %st(1), %st(0)`	`[A*B(A+C),D]`
`faddp %st(1), %st(0)`	`[A*B(A+C)+D]`
`fstp X`	`[]`

Example 3. Again suppose that the calculation $X = A * B(A + C) + D$ is to be performed, but for illustration purposes let us now assume that the floating point stack only has three slots. This means only two pseudo variables, `%f`$_0$ and `%f`$_1$ can be used since there might be need of a scratch slot. Instead spill slots denoted by `%sp(i)` are used where `i` is limited by the size of the native stack.

```
fmov A, %f0
fmov B, %f1
fmov C, %sp(0)
fmov D, %sp(1)
fadd %f0, %sp(0)
fmul %f0, %f1
fmul %f1, %sp(0)
fadd %sp(1), %sp(0)
fmov %sp(0), X
```

The strategy is to leave spill positions that are live out at a certain point on the stack and hope that the new value will not have to leave the stack on account of another spilled value wanting to take its place.

Instruction	Stack
`fld A`	`[A]`
`fld B`	`[B,A]`
`fld C`	`[C,B,A]`
`fstp %sp(0)`	`[B,A]`
`fld D`	`[D,B,A]`
`fstp %sp(1)`	`[B,A]`
`fld %sp(0)`	`[C,A,B]`
`fadd %st(0), %st(1)`	`[A+C,A,B]`
`fxch %st(1)`	`[A,A+C,B]`
`fmulp %st(2), %st(0)`	`[A+C,A*B]`
`fmulp %st(1), %st(0)`	`[A*B(A+C)]`
`fadd %sp(1)`	`[A*B(A+C)+D]`
`fstp X`	`[]`

Some notes on precision. The standard precision of floating point values in ERLANG is, as mentioned above, the IEEE double precision. On the x87, however, the precision is 80 bit double extended precision and whenever a floating point value of another type is loaded on the stack it is also converted to this precision.

When the bytecode is interpreted one instruction at a time, as it is in the BEAM interpreter, the operands are pushed to the stack and converted, the operation is performed, and finally the result is popped. The popping involves conversion back to the double precision by rounding the value on the stack.

When using the scheme described above, the results are kept on the x87 stack as long as possible if they are to be used again, which leads to a higher precision in the subsequent computations since no rounding is taking place in between computing an (intermediate) result and using it. This difference in precision can lead to different answers to the same sequence of FP computations depending on which scheme is used. The bigger the block of floating point instructions, the bigger the chance of getting different results. Note however that since less rounding leads to smaller accumulated error, the longer a value stays on the x87 stack, the better the FP precision that is obtained.

6 Performance Evaluation

Two points of interest are considered when evaluating the performance of floating point handling in ERLANG.

- How much does the compilation scheme described in this report affect the performance of ERLANG programs both when running in the BEAM interpreter and in native code?
- Does this scheme make Erlang/OTP competitive with state-of-the-art implementations of other strict functional languages in handling floating point arithmetic? Is the resulting performance as good as that of statically typed languages?

These questions are addressed below: In Section 6.1 the performance of the BEAM interpreter, HiPE, and SML/NJ are compared, followed by Section 6.2 which contains a performance comparison of different ERLANG implementations. The platforms used were a SUN Ultra 30 with a 296 MHz Sun UltraSPARC-II processor and 256 MB of RAM running Solaris 2.7, and a dual processor Intel Xeon 2.4 GHz machine with 1 GB of RAM and 512 KB of cache per processor running Red Hat Linux. Information about the ERLANG programs used as benchmarks can be found in Table 2.

6.1 Comparing floating point arithmetic in SML/NJ and Erlang/OTP

We have chosen to compare the resulting system against SML since it belongs to the same category of functional languages (strict) as ERLANG, it is known to have efficient implementations, and is statically typed so it can be seen how well the presented scheme performs against a system whose compiler has exact information about types and absolutely no type tests are performed during runtime. Note that this not restricted to floats but extends to *all* types. As such, it gives SML/NJ an advantage over Erlang/OTP, but

Table 2. Description of benchmark programs.

Benchmark	Lines	Description
float_bm	100	A small synthetic benchmark that tests all arithmetic floating point instructions; floating point variables have small live ranges.
float_bm_spill	100	Same as above but variables in the program are kept alive and spilling occurs.
barnes-hut	171	A floating point intensive multi-body simulator
fft	257	An implementation of the fast Fourier transform
raytracer	2898	A ray tracer that traces a scene with 11 objects (2 of them with textures)
pseudoknot	3310	Computes the 3D structure of a nucleic acid; programs are from [2]

provided that the benchmark programs are floating point intensive, one can expect that the manifestation of this advantage is not so profound.

Two versions if SML/NJ are being used. Version 110.0.7 is a stable, official realease of the compiler, but it is also a bit old. Thus the comparison has been extended to use a working version (110.41) of the compiler. Information about the SML/NJ compilers can be found at `cm.bell-labs.com/cm/cs/what/smlnj/`

Since SML/NJ generates native code, only a performance comparison against HiPE, which compiles floating point operations to native code using the scheme described in the previous sections, is presented. Table 3 contains the results of the comparison in three of the benchmarks. [1] **float_bm** shows the same picture on both SPARC and x86: SML/NJ version 110.41 is about 50% faster than HiPE on this program. On **float_bm_spill** the results are dependent on the platform: on the SPARC the difference is still about 50% but on the x86 the difference is down to 20%, something that speaks for the algorithm for mapping to the x87 stack. (Note is that this scheme is based on the same algorithm that SML/NJ version 110.41 is using.) We believe that this also validates the choice of the algorithm sketched in Example 3 for choosing which values to leave on the x87 floating point stack.

When it comes to **barnes-hut** it can be seen that the performance of SML/NJ has been improved in version 110.41. Compared to the older version, HiPE was only a few percent slower, but now the difference is a factor of up to 2.8 in the case of x86. As for **pseudoknot** the comparison is limited to the older version and there HiPE is even better on the x86 while 2.5 times worse on SPARC.

[1] Both versions of **float_bm** are small programs and so equivalent SML versions were written by the author; **raytracer** was too big a program to also be rewritten. **pseudoknot** and **barnes-hut** are standard benchmark programs of the SML distribution. The **fft** program typically used as an SML benchmark uses destructive updates and thus does not have the same complexity as the ERLANG one. **pseudoknot** could not be compiled by SML/NJ version 110.41.

Table 3. Performance comparison between HiPE and SML/NJ (times in ms).

Benchmark	HiPE	110.0.7	110.41
float_bm	4680	2660	2860
float_bm_spill	6320	4140	4190
barnes-hut	4540	4280	2180
pseudoknot	1530	610	—

(a) Performance on SPARC.

Benchmark	HiPE	110.0.7	110.41
float_bm	850	790	550
float_bm_spill	1620	1670	1360
barnes-hut	880	870	310
pseudoknot	140	190	—

(b) Performance on the x86.

6.2 Performance of float handling in implementations of ERLANG

In Erlang/OTP R9 the analysis described in this report and the floating point instructions are part of the BEAM compiler and interpreter. However, the compiler can be instructed not to use these so that all floating point arithmetic is performed using generic BEAM instructions operating on boxed values that have to be type tested and unboxed each time the value is used.

To study the performance of the presented scheme a comparison is made using Erlang/OTP R9 both with and without the floating point optimisations and finally using the HiPE compiler .

The results of the comparison are shown in Table 4. One can see that the performance of floating point manipulation in Erlang/OTP has improved considerably both as a result of using the analysis in the BEAM interpreter and due to the use of this information by the HiPE compiler. Note that the performance of e.g., **float_bm_spill** has improved up to 4.7 times by using the floating point instructions and the performance improvement due to native code compilation of floating point operations ranges from a few percent up to a factor of 3.6, again in the **float_bm_spill** program. It should be clear that the scheme described in this report is worth its while.

6.3 Performance of the static analysis in BEAM R9.

As can be seen in Table 5 the static analysis succeds in finding most of the floating point arithmetic instructions, but one thing to note is that this demands that the programmer is aware of the effect of guards. By adding guards in just a few of the functions in **barnes-hut** the percentage of discovered floating point arithmetic operations increased from 27% to 67%, wich in turn gave a speed-up of 25% on the x86. The performance on the different versions of **float_bm** is not surprising since they are of a synthethic

Table 4. Performance comparison between BEAM R7, R9, and HiPE (times in ms).

Benchmark	No fp opts	BEAM R9	HiPE
float_bm	39120	14800	4680
float_bm_spill	76640	23110	6320
barnes-hut	10890	10050	4540
pseudoknot	4850	2970	1530
raytracer	9260	9050	8290
fft	19600	16740	8890

(a) Performance on SPARC.

Benchmark	No fp opts	BEAM R9	HiPE
float_bm	9570	2460	850
float_bm_spill	17820	3750	1620
barnes-hut	2300	1860	940
pseudoknot	930	530	140
raytracer	1840	1750	1410
fft	3740	3120	1680

(b) Performance on x86.

nature, but considering that **pseudoknot** is a more realistic program, it is noteworthy that the analysis found all of the floating point arithmethics.

One current limitation when using guards to rise the use of floating point instructions is that no type information is propagated over basic blocks in BEAM. This means that if an argument is proved to be a float by a guard in the function head but is not used until after a branch (i.e., an if-statement) the operation will be treated as a general arithmetic operation after all.

Table 5. Performance of the static analysis for finding floating point arithmetic operations.

Benchmark	FP-operations	Discovered
float_bm	1×10^8	100%
float_bm_spill	2×10^9	100%
barnes-hut	1×10^8	67%
pseudoknot	8×10^7	100%
raytracer	3×10^7	79%
fft	8×10^7	94%

Acknowledgments

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

References

1. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Prog. Lang. Syst.*, 13(4):451–490, Oct. 1991.
2. P. H. Hartel et al. Benchmarking imlementations of functional languages with “pseudoknot”, a float intensive program. *Journal of Functional Programming*, 6(4):621–655, July 1996.
3. E. Johansson and K. Sagonas. Linear scan register allocation in a high performance Erlang compiler. In *Practical Applications of Declarative Languages: Proceedings of the PADL’2002 Symposium*, number 2257 in LNCS, pages 299–317. Springer, Jan. 2002.
4. A. Leung and L. George. Some notes on the new MLRISC x86 floating point code generator (draft). Unpublished technical report available from: http://cm.bell-labs.com/cm/cs/what/smlnj/compiler-notes/.
5. S. S. Muchnick. *Advanced Compiler Design & Implementation*. Morgan Kaufman Publishers, San Fransisco, CA, 1997.
6. M. Pettersson. A staged tag scheme for Erlang. Technical Report 029, Information Technology Department, Uppsala University, Nov. 2000.
7. M. Poletto and V. Sarkar. Linear scan register allocation. *ACM Trans. Prog. Lang. Syst.*, 21(5):895–913, Sept. 1999.