Uppsala Master's Theses in
Computing Science 100
Examensarbete DV3
October 2, 1996
ISSN 1100–1836

Native Code Compilation for Erlang

Erik Johansson & Christer Jonsson

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

Examiner: Håkan Millroth

Passed:

Abstract

We describe the design and implementation of a native compiler for ERLANG, built on top of an existing emulated implementation. Many ERLANG-programs are time-critical, and as ERLANG becomes more widely used, the need for fast implementations increase. We show how simple native compilation can increase the speed of ERLANG programs, even compared to ERLANG programs compiled via C. Several benchmark programs are examined and the results analysed, showing that our simple method gives a considerable gain in execution speed over current implementations.

Contents

1 Introduction **1**

2 JAM **3**
2.1 Registers . . . 3
2.2 Data Representation . . . 4
2.3 Code . . . 4
2.4 Calls . . . 4
2.5 Concurrency . . . 5

3 The Compiler **6**
3.1 Translation . . . 7
3.1.1 Large Instructions . . . 7
3.1.2 Failure . . . 8
3.2 Optimization . . . 9
3.2.1 Constant propagation . . . 9
3.2.2 Constant folding . . . 10
3.2.3 Unreachable code elimination . . . 10
3.2.4 Dead code removal . . . 10
3.2.5 An Example . . . 11
3.3 Machine specific transformations . . . 11
3.3.1 Delay-slot filling . . . 12
3.4 Register Allocation . . . 12

4 System Integration **13**
4.1 Function Information . . . 13
4.2 Calls and Return . . . 13
4.3 Hot Code Loading . . . 15
4.4 Adaptive Compilation . . . 15

5 Performance Evaluation **17**
5.1 Effects of Optimizations . . . 19
5.2 Code Size . . . 20
5.3 Compilation Speed . . . 20

6 Conclusion & Future Work **21**

A The JAM Instruction set **25**

B Translation scheme **29**

List of Figures

2.1 Representation of the term {`42, [foo, bar]`}. 4
2.2 The stack after a call. 5

3.1 An ERLANG function that returns 1 when called with the empty list, and 0 otherwise. 8
3.2 The JAM code for the function in figure 3.1. 9
3.3 The native code for the function in figure 3.1. The `RETURN` instruction actually expands to seven machine instructions. . 9
3.4 An ERLANG function calculating the length of a list. 11
3.5 Optimizing the function length. 11

4.1 A call and return from a native function to a JAM-function . 15

5.1 A comparison of execution times 18
5.2 Optimization gains 19

List of Tables

5.1 Relative execution times 19

B.1 Stack instructions 29
B.2 Test Instructions 30
B.3 Process instructions 30
B.4 Arithmetic instructions 31
B.5 Call instructions 32

Chapter 1

Introduction

In this paper we will present a straightforward implementation of a native code compiler for ERLANG[4]. ERLANG is a functional programming language intended for use in telecommunication switches with high demands on availability. This intention has led to the demand that ERLANG object code should be small, and that it should be replaceable at runtime (hot code loading). These requirements makes it natural to implement ERLANG using a byte code emulator. Since it is hard to implement a fast emulator, other approaches are desirable. One approach is to compile ERLANG into C, and then compiling the C code into native code. Good optimizing compilers are widely available for C, making it possible to generate efficient and portable code. However, there are drawbacks to such an approach:

- Many ERLANG features, such as tail-call optimization and hot code loading, are difficult to implement in C.
- Information is lost in the translation to C, making some optimisation impossible, for the C compiler.
- An extra (often expensive) compilation step is required.
- The resulting object code is large.

We think that these limitations can be avoided, at the cost of portability, by compiling directly to native code.

Approach

We wanted to test this hypothesis, without having to write a complete ERLANG system. By building our compiler on top of JAM (see chapter 2) we got a complete system up and running, in less than six months. The compiler is implemented as a runtime compiler that produces native code for the SPARC architecture.

Runtime compilation opens the way for numerous optimization techniques. We have not explored these yet, and therefore this paper concentrates on native compilation as such, and not on runtime compilation. However, we plan to extend our compiler with runtime optimizations.

Our main interest lies in examining the possible gains of native compilation, and not in general ways of improving upon previous ERLANG implementations. Therefore we have not made any changes to the rest of the runtime system. To clearly see the effect of native compilation, we have concentrated our work on sequential ERLANG code.

Related Work

Compilation of ERLANG to C has been implemented by Hausman in his BEAM compiler [12], and we have compared our results with this compiler. Inspiration to our work comes from many sources:

- Ertl [10] has inspired us with his work on compiling stack machines.
- Haygood [11] has described native compilation for SICStus Prolog.
- Runtime compilation has been examined by several researchers, for example Chambers [9], and Leone and Lee [16, 15].

Results

On the average, our compiler is 55% faster than ERLANG compiled via C, and more than four times faster than emulated ERLANG. Our generated code is about 60% smaller than ERLANG compiled via C. The compiler generates code from byte code to native code an order of magnitude faster than GCC compiles from C to object code.

Bear in mind that these results are for sequential code, and that for typical ERLANG programs, utilizing processes and message passing, the overall performance increase might not be as large.

Chapter 2

JAM

We have built our compiler on top of JAM [3], which is a stack based byte code emulator for ERLANG. ERLANG is a dynamically typed functional programming language, with built in support for processes and communication. No destructive updates of variables and data structures are allowed, and memory is managed automatically. The language has been inspired by ML, Prolog, and other declarative languages, and it supports for example pattern matching.

Joes Abstract Machine, or JAM, is named after its first implementator Joe Armstrong, who also is one of the chief architects behind ERLANG. In this chapter we survey the inner workings of JAM.

2.1 Registers

JAM uses the following registers:

`PC` - Pointer to the next instruction to execute.

`STOP` - Pointer to the stack top of the active process.

`HTOP` - Pointer to the heap top of the active process.

`ARGS` - Pointer to the first argument (on the stack) of the current function.

`VARS` - Pointer to the first local variable on the stack.

`FAIL_PC` - Address where execution should continue after a failure.

`FAIL_REASON` - The reason for the failure.

`P` - Pointer to the process control block (PCB).

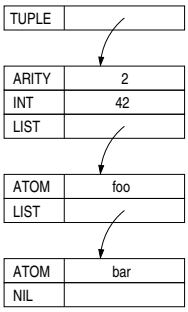


Figure 2.1: Representation of the term {`42, [foo, bar]`}.

2.2 Data Representation

All basic values are represented in one machine word (in this case 32 bits). A tag is stored in the four most significant bits, leaving 28 bits for the value. Small integers, atoms and nil (the empty list) can be stored directly in a machine word. For more complex values, such as lists and tuples, a pointer to a heap-allocated object is stored in the word.

A list cell on the heap is just two consecutive words. Tuples start with a header containing the arity of the tuple, followed by its elements.

2.3 Code

The JAM instruction set (described in Appendix A) is implemented with byte codes, making it compact. Code is patched at load time (for example with indexes into the atom table). A module in ERLANG is a collection of functions sharing the same name space. Code is loaded one module at the time, when needed.

2.4 Calls

Before entering a function, the function arguments are pushed on the stack. Then a stack frame is written, containing the return address, a pointer to the code of the calling function (`CC`), and the old values of `ARGS` and `VARS` (see figure 2.2). `ARGS` is set to point to the first argument, and `VARS` to point to the first free stack position. When the function returns, the frame is popped from the stack and the return value is pushed (in the same stack position as the first argument).

The JAM compiler recognizes tail calls, that is, calls that are the last instruction in a function. Before a tail call the current stack frame can be freed, making it possible to execute loops in constant stack space. This is called tail call optimization or last call optimization.

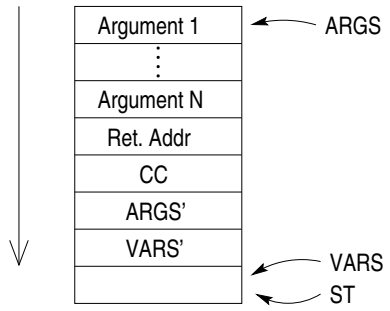


Figure 2.2: The stack after a call.

According to the specification of ERLANG, there are two types of calls, local calls (calls within a module,) and remote calls (calls to functions in other modules). Since all functions in a module are loaded at the same time the address of a local call can be determined at compile time. The address of a remote call on the other hand, can change at runtime and the emulator performs a table lookup for each remote call.

2.5 Concurrency

A prominent feature of ERLANG is its ability to handle many lightweight processes and to supply constructs for message passing between these processes.

In JAM, each process has its own process control block (PCB), stack, and heap. When a process is not executing the registers of the abstract machine are stored in the PCB.

A scheduler queue is maintained, containing processes awaiting execution. These processes are scheduled on a round robin basis. A process executes until it either waits for a message, or has consumed its time slice. The time slice is set to a number of reductions (function calls). When a process has used up its time slice it is swapped out, and placed last in the scheduler queue.

Message passing is implemented by the sending process copying the message into the heap of the receiving process. After the message is written, a pointer to the message is inserted into the message queue of the receiving process and the receiving process is added to the scheduler queue (if it is not there already).

Chapter 3

The Compiler

Our compiler compiles from JAM code instead of compiling from ERLANG source code. This might seem as a disadvantage at first, since higher level optimizations will be hard to implement. But this approach has several advantages:

- **Compilation on a function basis.** The compiler does not have to compile a complete module, but can compile each function separately, saving space and compilation time.
- **No source code necessary.** Since the compilation starts from JAM code, the ERLANG source code need not be available. Functions in the standard libraries can be compiled to native code[1].
- **Easy compilation.** The JAM code is easier to compile than ordinary ERLANG code.

The compilation consists of these stages:

Translation. In this stage, the JAM code is translated into intermediate-code. We use techniques worked out by Ertl [10] to translate stack based JAM code to register code.

Optimization. After the translation the compiler make some global (in the sense of reaching over the whole function) transformations, performing some standard optimizations, such as constant propagation and dead code elimination.

Machine specific transformations. At this stage the code is adapted for the target architecture, in this case the SPARC.

[1] You would not believe the gain in speed that can be achieved by compiling the lists module.

Register Allocation. The process of deciding which register to keep a value in is called *register allocation*. It is desirable that as many values as possible are held in registers for their whole lifetime. Graph coloring register allocation as described by Chaitin et al. [7, 8] is used to achieve this.

Assembly. In the final stage of the compilation the size of the generated code is calculated, memory is allocated, and the actual machine instructions are written to memory.

3.1 Translation

Since SPARC is a register machine, it is desirable to use the JAM stack as little as possible. There are several possible solutions to this problem. One approach would be to keep the stack as it is and by means of analysis, keep the most used stack positions in registers.

We have chosen another approach. Inspired by Ertl's [10] work on Forth, we wanted to transform all stack references to register references in the translation stage.

A virtual stack is maintained during translation, corresponding to what the real stack would look like at runtime. By assigning registers to each position in this virtual stack, each temporary value on the JAM stack is assigned a register, resulting in fewer memory references. Stack operations like `pop` and `alloc` only affect the virtual stack at compile time, and do not generate any code.

3.1.1 Large Instructions

Some large instructions (such as `send`) are still executed in the emulator to avoid code explosion. These are mainly instructions that already have large overheads. Another approach would be to create a library of these functions in native code. In fact, that is what we have done for some of the arithmetic instructions.

The types of the arguments to an arithmetic expression in ERLANG can be either one of integer[2], float, or bignum[3]. Generating native code for all possible combinations induces severe code bloat. We have opted for a solution where integer addition and subtraction is done inline. Floating point arithmetic is done with calls to machine code routines. Calls to these routines are cheaper than calls to the emulator since they share the same execution environment as our native code. The rest of the arithmetic uses the original code in the emulator.

[2] *i. e.*, fixed precision integers.
[3] *i. e.*, Arbitrary precision numbers.

```
is_nil([]) -> 1;
is_nil(_) -> 0
```

Figure 3.1: An ERLANG function that returns 1 when called with the empty list, and 0 otherwise.

All built in functions (bifs) are executed by the emulator, although some bifs probably could be inlined. As an optimization, our compiler determines the address at compile time (in contrast to JAM, which determines the address of the bif at execution time).

3.1.2 Failure

In ERLANG, choice and flow of control is done by pattern matching. Each available choice is tested until a match occurs. This is achieved by use of the instructions `try_me_else`, `try_me_else_fail`, `commit`, and test instructions that can succeed or fail. Consider the function `is_nil/1` in figure 3.1. This function will be compiled to JAM code as follows. The first instruction of the first clause will be the instruction `try_me_else(label1)`, indicating that there is another clause to try if a failure should occur while matching the first clause.

Then the argument to the function has to be tested, to examine if it is `nil`. Therefore the instructions `arg(0)` and `getNil` are generated, to push the first argument on the stack and to test if the top of the stack contains `nil`. If `getNil` fails (because the stack top does not contain `nil`) then the execution will continue at `label1`. Otherwise execution will continue with the next instruction. Since the function head has been matched, the execution is now commited to the first clause.

After the `commit` instruction, code for the body of the first function clause will be generated. Giving the instructions `push_int(1)` and `ret`, indicating that the function should return with the value 1.

Then the second clause will be compiled. This clause will begin with the label `label1`. After the label, a `try_me_else_fail` instruction will be generated, indicating that if the function head cannot be matched now, then the process should fail.

Since the second clause accepts any argument, no code will be generated to test the arguments. Instead a new `commit` instruction and the code for the body of this clause is generated. The complete JAM code for the function can be found in figure 3.2.

In our compiler the location of each label is determined at compile time and test instructions that will not generate a real fail, are compiled to branches to the corresponding label. The native code for the function can be found in figure 3.3.

```
            try_me_else(label1)
            arg(0)
            getNil
            commit
            pushInt(1)
            ret
label1:     try_me_else_fail
            commit
            pushInt(0)
            ret
```

Figure 3.2: The JAM code for the function in figure 3.1.

```
      r1 := VARS[-20]
      if r1 != nil then L2
      r1 := 0x10000001
      RETURN(r1)
L2:   r1 := 0x10000000
      RETURN(r1)
```

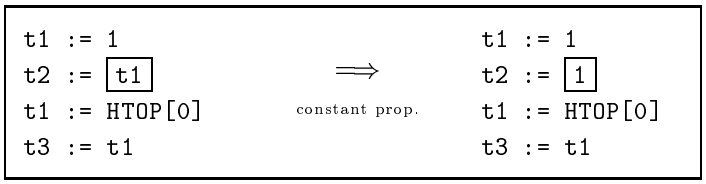
Figure 3.3: The native code for the function in figure 3.1. The `RETURN` instruction actually expands to seven machine instructions.

3.2 Optimization

All our optimizations depend on *global data-flow analysis*. Data-flow information is collected by setting up and solving systems of equations [1, ch. 10].

3.2.1 Constant propagation

If an instruction assigns a constant value to a register, this value is propagated through the code, and used instead of the register in all instructions between this instruction and the first instruction that writes to this register.



3.2.2 Constant folding

If an instruction performs arithmetic on two constants, the result is computed and the instruction is replaced by an instruction that assigns the result to the destination register.

```
t1 := [1 + 2]   ⟹ (Constant folding)   t1 := [3]
```

If a branch instruction tests two constants, this instruction is replaced by a jump instruction or removed, depending on the result of the test.

```
[if 1 < 2] goto L1   ⟹ (Constant folding)   goto L1
```

3.2.3 Unreachable code elimination

Basic blocks that have no predecessors are removed.

```
goto L1
[t1 := 1]
L1:
t2 := t1
```

⟹ (Unreachable code)

```
goto L1
L1:
t2 := t1
```

3.2.4 Dead code removal

If a value is assigned to a register, and that register is not used after the assignment, then the assignment is removed.

```
[t1 := 1]
t1 := 2
```

⟹ (Dead code removal)

```
t1 := 2
```

```
len([]) -> 0;
len([_|L]) -> 1 + len(L).
```

Figure 3.4: An ERLANG function calculating the length of a list.

3.2.5 An Example

Since ERLANG is dynamically typed, the JAM-code will contain a lot of type tests. These tests can sometimes be removed by our optimizations. When the ERLANG function in figure 3.4 is translated, the code in figure 3.5 will be generated for the addition.

After register `r5` is assigned the tagged integer 1, a recursive call to `len` is executed (not shown) and the return value will end up in register `r9`. Both registers are tested if they contains integers; if not, a more complex routine is called (not shown). The tags are then removed from `r5` and `r9`, they are sign extended, added and the result is placed in `r15`.

After constant propagation and constant folding, the test whether `r5` is an integer (5, 6) is constant, and can be removed. In the same way the tagging and the sign-extending of `r5` (8, 10) are constant and can be removed. Now the constant 1 can be added directly to `r9`.

This leaves us with only four arithmetic operations and one conditional branch, a marked improvement on the original seven arithmetic operations, two conditional branches, and one assignment.

```
 1:  r5 := 0x10000001
 2:  ...
 3:  r11 := r9 >> 0x1c                ...
 4:  if r11 != 0x1 goto L13           r11 := r9 >> 0x1c
 5:  r12 := r5 >> 0x1c                if r11 != 0x1 goto L13
 6:  if r12 != 0x1 goto L13     ⟹    r9 := r9 << 0x4
 7:  r9 := r9 << 0x4                  r9 := r9 >>?  0x4
 8:  r5 := r5 << 0x4                  r15 := [0x1] + r9
 9:  r9 := r9 >>?  0x4
10:  r5 := r5 >>?  0x4
11:  r15 := [r5] + r9
```

Figure 3.5: Optimizing the function length.

3.3 Machine specific transformations

After the optimization phase, the compiler adapts the code to the actual machine, in this case the SPARC. Most SPARC instructions allow only 13-bit immediates, meaning that instructions with larger immediates must be

split into three instructions: two instructions that assigns the immediate to new temporary register, and the original instruction using this register instead of the immediate.

3.3.1 Delay-slot filling

In the SPARC architecture, there is a delay-slot after each branch instruction that can be executed even if the branch is taken. There are considerable performance gains to make if these delay slots can be filled with useful instructions. We do not do a state-of-the-art branch instruction scheduling, but search the basic block preceding the branch for a suitable instruction [13].

3.4 Register Allocation

The registers in our intermediate code must be assigned to machine registers. We use pessimistic graph coloring register allocation [5, 7, 8]. This is an abstraction of the register allocation problem to a graph coloring problem. Graph coloring is NP-complete so a heuristic method is required. Our implementation of the algorithm consists of 4 phases:

1. Calculate liveness. We have to know which registers are alive at a specific point in the program. This is done by global dataflow analysis [1].

2. The liveness information is used to build an interference graph where the nodes are registers and an edge between two nodes indicates that they are simultaneously live.

3. There are K available machine registers, this graph has to be colored with K colors. Locate a node with fewer than K neighbours and push it on a stack. As this node has at most K-1 neighbours, a free color can always be found. Remove this node, and its edges, from the graph and repeat this phase until the graph is empty or until no nodes with degree $<$ K can be found.

4. The registers are popped from the stack, inserted back into the graph and given a color that none of its neighbours got.

We have not changed the runtime system to be able to handle spilled values on the stack. If the graph cannot be colored, the compilation of the function fails and the original emulated code is used. This rarely happens (it never happened in our benchmarks).

Chapter 4

System Integration

The compiler is implemented as a runtime compiler, integrated with JAM. This forces the system to be able to find out at runtime if a function is compiled to native code or not. We will describe how this is solved, along with some other aspects of the integration between our compiler and the JAM emulator.

4.1 Function Information

When the JAM code is loaded the compiler adds a header with information to each function. A field in the header indicates whether the function has been compiled or not, and where the compiled code resides. All headers are linked together in a list.

4.2 Calls and Return

To speed up calls, we have replaced the `CC` field in the stack frame with a field for the `NPC` (Native Program Counter). The `CC` field can be removed since the information stored in it can be found by searching through the list of loaded functions. This search is only needed after a failure, therefore the speed is not as important as in a call.

We have also added four named addresses to the emulator; these are the `emulator_entry_point`, the `emulator_return_point`, the `native_entry_point`, and the `native_return_point`. When control passes between emulated code and native code, it passes through one of these points. Otherwise the implementation is simple: calls and returns from emulated code to emulated code are analogous to the original JAM, while calls and returns from native code are ordinary jumps.

Let us therefore look at calls and returns between emulated and native code.

- **Call from emulated to native.** If the information in the header of the called function indicates that there is a native version of the function then a call to native code will be executed.

  A call from emulated to native starts out the same way as a call from emulated to emulated, the stack frame is written in the same way. But after that the execution goes via the `native_entry_point` to the native code. The address of the called function is found in the header of the JAM function.

- **Call from native to emulated.** A call from native code is executed in the same way regardless of the destination. The value `null` is written to the `PC` field of the stack frame to indicate that the call originated from native code. The address of the called function is always available as an immediate argument to the call instruction.

  If the called function is not native compiled, then the call passes through a stub in the header of the JAM function. In this stub the `PC` register is set to point to the address of the JAM function, whereafter execution enters the emulator via the `emulator_entry_point`.

- **Return from emulated to native.**

  When the emulator reaches a `ret` instruction it checks whether the `PC` field in the stack frame is `null` or not. If it contains an address then the function returns to an emulated function and execution in the emulator goes on as usual.

  If the `PC` field is `null` then the call originated from native code. Therefore `NPC` is reset to the value of the `NPC` field in the stack frame, and execution reenters native code via the `emulator_return_point`.

- **Return from native to emulated.** In native code a return always goes to the address saved in the `NPC` field. If the call came from native code then this address will point to native code, and the return will be a swift one.

  If, on the other hand, the call came from emulated code then the `NPC` field contains a pointer to the `native_return_point`, and execution will continue in the emulator. In both cases `PC` has been restored to the value in the `PC` field by the returning function.

With this scheme native-to-native calls and returns are fast; we do not have to do any runtime tests whatsoever. Calls between emulated and native (and back) are a little bit slower since there is an extra indirection involved. But if a sufficent number of the frequently executed functions are compiled, then there will be few transistions between native code and emulated code.

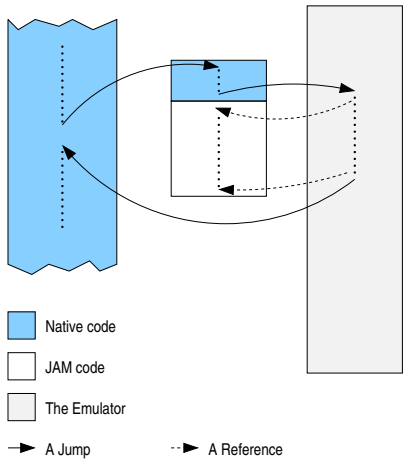


Figure 4.1: A call and return from a native function to a JAM-function

4.3 Hot Code Loading

As mentioned earlier ERLANG supports *hot code loading*: a code module can be replaced while the program is running. This means that the address of a function can change at runtime.

In JAM this is solved by determining the destination of a remote call at runtime. In our native code compiler the problem is solved with backpatching.

The solution is straightforward. When a function is replaced by new code all remote callers are patched to call the new emulated function. Since the old function should not be backpatched any more, all references to the old function are removed from all backpatch lists.

With this approach we achieve higher execution speeds, at the cost of marginally slower code loading.

4.4 Adaptive Compilation

The fact that the compiler is a *runtime compiler* opens up the possibility of letting the compiler choose which functions to compile to native code and wich to leave as byte code. We call this process adaptive compilation [14].

Adaptive compilation is achieved by counting the number of calls to each function. When the counter reaches a limit (C), the function is compiled.

Over time, those functions that are called most will be compiled. As to not let every function be compiled, all call counters are reset after some fixed time T. This way only the most frequently called functions are compiled.

With compilation based on a per function basis, as opposed to a per module bases, we can keep the code size down, and still gain in performance.

By varying the C and T parameters the same compiler can work in different environments (space critical or time critical). We have not investigated this further, but much work in this direction has been done on SELF by Hölzle [14].

There are a number of analyses that a runtime compiler can do, that are hard for a static compiler, but we have not explored this field yet. We will discuss this further in chapter 6.

Chapter 5

Performance Evaluation

We have compared our implementation with emulated JAM 4.3.1, BEAM/T 4.3 and BEAM/C 4.3. BEAM [12] is a register machine originally designed to be compiled to C. The current implementation supports a direct threaded emulator (BEAM/T) and compilation via C (BEAM/C).

We have used nine sequential benchmarks to evaluate our compiler. The size of the benchmarks varied between three lines (fibonacci and length) and over 1000 lines (raytracer). The measurements were made on a Sun SPARCsystem 600-4 with 128 megabytes of memory running SunOS 5.5. Each benchmark runs for approximately 1-2 minutes and was run 3 times giving a total running time of well over 30 minutes. The benchmarks used were:

Huffman. A huffman encoder. Compresses and uncompresses a text file of 32 kilobytes. 138 lines.

Smith-Waterman. The Smith-Waterman DNA sequence matching algorithm [18]. Matches a sequence of length 32 to 600 other sequences of length 32. 68 lines.

Barnes-Hut. Simulates gravitational forces between 1000 bodies. 156 lines.

Raytracer. A ray tracer. Traces a picture with spheres, planes and texture mapping. Approximately 1000 lines.

Quicksort. Ordinary quicksort. Sorts a list of 15000 random integers 20 times. 16 lines.

Fibonacci. A recursive fibonacci. Calculates fib(30) 10 times. 3 lines.

Length. A tail recursive list length function. Takes the length of a 200000 element list 50 times. 3 lines.

Naive reverse. Naive reverse of a 1000 element list 20 times. 8 lines.

Tak. Takeuchi function, recursive arithmetic. Calculates tak(18, 12, 6) 250 times. 9 lines.

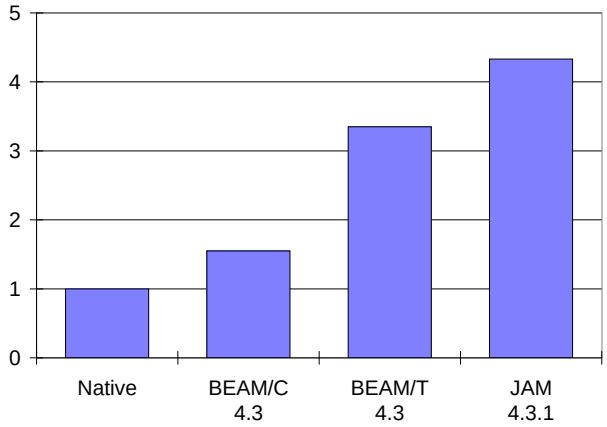


Figure 5.1: A comparison of execution times

Results

On the average our compiler is 55% faster than BEAM/C (60% when excluding garbage collection time). To get a more detailed picture we take a look at the benchmarks partitioned into three classes:

Integer. Huffman, Smith-Waterman, Fibonacci, Length, and Tak perform a lot of integer arithmetic. This is where our compiler does best, on the average over 2 times faster than BEAM/C. The main reason is that our compiler does integer arithmetic inline while BEAM/C, mostly compiles arithmetic operations into function calls.

Floating point. Barnes-Hut and the raytracer use floating point arithmetic extensively. Both BEAM/C and our compiler perform a subroutine call for floating point arithmetic. Our compiler is 20% faster on these benchmarks.

List processing. Naive reverse and quicksort We are 6% faster on quicksort but 27% slower running naive reverse. The reason for this is that BEAM compiles the function append into a very tight inner loop.

Our compiler gets a speedup of more than four times compared to JAM, most of which (90%) stems from removal of emulation overhead and avoiding use of the stack. The remaining 10% comes from passing arguments

in registers and optimizations made possible when JAM instructions are merged into larger pieces of code. As compared to BEAM, and for the same reason, our compiler is not so much faster than JAM on the floating-point benchmarks.

Our garbage collection times are 56% less than for the JAM, even though we use the same garbage collector. We believe that by keeping arguments and temporary values in registers, and only saving live registers to the stack, we allow the garbage collector to reclaim more memory.

Benchmark	**Native**	**Jam 4.3.1**	**BEAM/T 4.3**	**BEAM/C 4.3**
Huffman	1.00	9.03	4.45	2.31
Smith-Waterman	1.00	4.94	3.77	1.86
Fibonacci	1.00	4.40	6.24	1.84
Length	1.00	8.51	5.58	2.13
Tak	1.00	4.88	4.87	2.50
Barnes-Hut	1.00	1.42	1.81	1.23
Raytracer	1.00	1.77	1.72	1.17
Naive reverse	1.00	5.99	1.82	0.79
Quicksort	1.00	4.36	1.92	1.06
	1.00	4.33	3.35	1.55

Table 5.1: Relative execution times

5.1 Effects of Optimizations

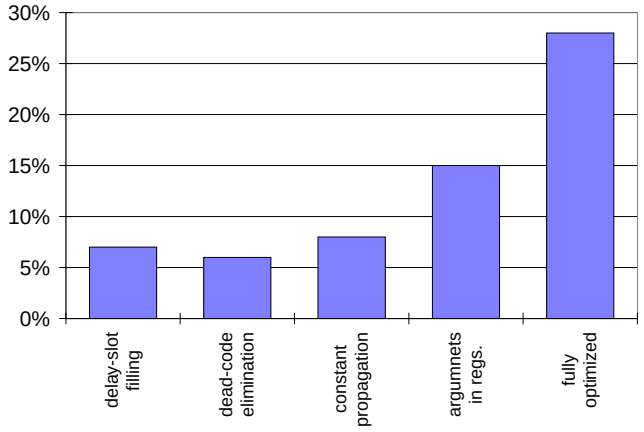


Figure 5.2: Optimization gains

In figure 5.2 we display the effects of our optimizations. We switched off each optimization, one at a time, and compared the execution times with the non-optimizing compiler and with the fully optimizating compiler.

Constant propagation, unreachable code elimination, and constant folding are all done in one pass and cannot be run individually. They are all accounted for under the label constant propagation. In total, all the optimizations give a useful 28% speedup.

5.2 Code Size

The size of the benchmarks compiled with our compiler grows from 9 kilobytes of JAM code to 196 kilobytes of native code. The code for the same benchmarks for BEAM/C is 315 kilobytes.

It can be argued that the size of our code is much bigger than JAM and not significantly smaller than the one generated by BEAM/C; but we believe that by having the runtime system compile only the most frequently executed functions, the average code size will not increase much, and the gain in speed will be almost the same.

5.3 Compilation Speed

Our compiler compiled the benchmarks in 34 seconds (45 seconds including Erlang to JAM compilation). BEAM/C compiled the benchmarks in 447 seconds (464 seconds including Erlang to BEAM compilation).

Chapter 6

Conclusion & Future Work

Our results show that our compiler generates native code which is more than four times faster than JAM and 55% faster than BEAM compiled via C. These results are achieved without any optimizations on ERLANG code, without any global optimizations over functions boundaries, without inlining, without instruction scheduling, without (major) changes to the runtime system, and without a type inference system. There is probably a lot to be gained from a further refined system.

It is not more complicated to compile into native code than into C. One achieves greater control over the final code and specific information about the language and runtime system are not lost in the process. Apart from portability, there are no real reasons to compile ERLANG via C.

Future work.

One aspect that we have not fully examined, is which optimizations we could do by using information available only at runtime. Examples of possible optimizations are:

Specializing functions by their actual arguments. By looking at the actual values that functions are called with, these functions could be specialized with respect to the types or maybe even the values themselves [14, 16, 15].

Ordering function clauses after use. The function clause that is taken most often should be tried first. Most ERLANG programmers knows this and places, for example, the test for the empty list last. But it is not always trivial to see beforehand which clause to place first. The compiler can profile running code to infer this, and rearrange the code accordingly.

Tuning branch instructions. A similar optimization on a lower level is to profile each branch instruction. Then the code could be rearranged

in order to make branch prediction easier for the processor [6].

Storing often executed basic blocks together. The profiling information generated (at runtime) by the above mentioned techniques could also be used to find the most frequently executed basic blocks. These could then be stored together to improve instruction cache behavior [17].

We plan to examine these possibilities, and at the same time try to find other possible techniques. We also plan to examine the behavior of adaptive compilation, to see how much faster a system can get with limited increase in code size, by letting the compiler decide which functions to compile.

Bibliography

[1] Aho, V. A., Sethi, R. and Ullman, J. D., *Compilers: Principles, Techniques, and Tools*. Addison-Wesley 1986.

[2] Aït-Kaci, H., *Warren's Abstract Machine*, MIT Press, 1991.

[3] Armstrong, J. L., Däcker, B. O., Virding, S. R. and Williams, M. C., *Implementing a Functional Language for Highly Parallel Real Time Applications*. SETSS 92, 30th March to 1st April 1992, Florence.

[4] Armstrong, J. L., Virding, S. R., Wikström, C., and Williams, M. C., *Concurrent Programming in Erlang*. Prentice Hall, second edition, 1996.

[5] Briggs, P., Cooper, K. D., and Torczon, L., *Improvements to graph coloring register allocation*. ACM transactions on programming languages and systems 16, 3 (May 1994), pp. 428-455.

[6] Calder, B., Grünwald, D., *Reducing Branch Costs via Branch Alignment*. Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV), April 1991. pp. 242-251.

[7] Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M.E., and Markstein, P. W., *Register allocation via coloring*. Computer Languages 6, (January 1981), pp. 47-57.

[8] Chaitin, G. J., *Register allocation and spilling via graph coloring*. SIGPLAN Notices 17, 6 (June 1982), pp. 98-105. Proceeding of the ACM SIGPLAN'82 Symposium on Compiler Construction.

[9] Chambers, C., *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*. PhD thesis, Stanford University, Stanford, California, March 1992. Tech Report STAN-CS-92-1420.

[10] Ertl, M. A., *A New Approach to Forth Native Code Generation*. EuroForth '92, pp. 73-78.

[11] HAYGOOD, R. C., *Native Code Compilation in SICStus Prolog*. International Conference on Logic Programming 1994, pp. 191-204. MIT Press, 1994.

[12] HAUSMAN, B., *Turbo Erlang: Approaching the Speed of C*. Implementations of Logic Programming Systems, pp. 119-135, Kluwer Academic Publishers, 1994.

[13] HENNESSY, J., L., PATTERSON, D., A., *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann Publishers, 1990.

[14] HÖLZLE, U., *Adaptive Optimization for Self: Reconciling High Performance with Exploratory Programming*. Ph.D. thesis, Computer Science Department, Stanford University, August 1994.

[15] LEE, P. AND LEONE, M., *Optimizing ML with Run-Time Code Generation*. ACM SIGPLAN '96 Conference on Programming Langauge Design and Implementation, 1996.

[16] LEONE, M. AND LEE, P., *Lightweight Run-Time Code Generation*. Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, pp. 97-106, June 1994.

[17] MCFARLING, S., *Procedure Merging with Instruction Caches*. ACM SIGPLAN '91 Conference on Programming Langauge Design and Implementation, 1991. pp. 71-79.

[18] SMITH, T. F., AND WATERMAN, M. S., *Identification of common molecular subsequences*. Journal of Molecular Biology 147, 1981, pp. 195-197.

[19] *The SPARC Architecture Manual, Version 8*, Prentice Hall, 1992.

Appendix A

The JAM Instruction set

alloc_{n}.[1] ($0 \leq n \leq 15$) Allocate space for n variables.

allocN(N). Allocate space for N variables.

apply_call. Call the function indicated by the top of the stack.

apply_enter. Perform a tail-optimised call to the function indicated by the top of the stack.

arg_{n}.[1] ($0 \leq n \leq 15$) Push argument n on the stack.

argN(N). Push argument N on the stack.

arith_{plus,minus,times,div}. Arithmetic operations.

arith_{band,bor,bxor,bnot,bsl,bsr,rem,intdiv,neg}. Bitwise arithmetic (on integers only).

bif_call(Bif). Call the bif Bif.

bif_enter(Bif). Call the bif Bif, and then return.

call_local(*Arity,Offset*). Call the function of arity *Arity* at `PC` + *Offset*.

call_remote($Arity, Function$). Call the remote function of arity *Arity* by looking in the export table for $Function$.

commit. Set `FAIL_PC` to `NULL` and `FAIL_REASON` to "Bad match".

comp_{gt,lt,geq,leq}. Compare the two top stack positions.

comp_{eqeq,neq}. Test for equality of the top two stack positions.

die. Kill the process.

[1] 16 different instructions.

dup. Duplicate the top of stack.

enter_local(*Arity, Offset*). Perform a tail-optimised call to the local function of arity *Arity* at `PC` + *Offset*.

enter_remote($Arity, Function$**).** Perform a tail-optimised call to the remote function of arity $Arity$, by looking in the export table for $Function$.

eqArg_{**n**}.[1] $(0 \leq n \leq 15)$ Fail if argument n is not equal to the top of the stack.

eqArgN(N**).** Fail if argument N is not equal to the top of the stack.

eqVar_{**n**}.[1] $(0 \leq n \leq 15)$ Fail if variable n is not equal to the top of the stack.

eqVarN(N**).** Fail if variable N is not equal to the top of the stack.

failCase. Fail with reason "case clause".

failIf. Fail with reason "if clause".

getAtom(A**).** Check that the top of the stack contains the atom A.

getFloat(F**).** Checks that the top of the stack contains the float F.

getInt_{**n**}.[1] $(0 \leq n \leq 15)$ Check that the top of the stack contains the integer n.

getInt1(N**).** Check that the top of the stack contains the integer N ($N < 256$).

getInt4(N**).** Check that the top of the stack contains the integer N.

getIntN($N, D1, D2, ..., DN$**).** Check that the top of the stack contains the (bignum) integer with the (16 bit) digits $D1$ to DN.

getNil. Check that the top of the stack contains the empty list nil.

getStr($N, E1, E2, ..., EN$**).** Match the top of the stack with a list of length N, containing the elements $E1$ to EN.

goto(*Offset*). Set `PC` to `PC` + *Offset*.

hash(H**).** Hash the argument on the stack with the hash value H.

head. Take the head of the list on the stack.

heap_need(N**).** Make sure that there is space for N word on the heap.

list_length. Calculate the length of the (well formed) list on the stack.

mkList. Make a list of the two top positions of the stack.

mkTuple_{n}.[1] ($0 \leq n \leq 15$) Make a tuple of size n, of the n top positions of the stack.

mkTupleN(N). Make a tuple of size N, of the N top positions of the stack.

pop. Pop.

popCatch. Remove the catch on the top of the stack.

popCommit. Do a pop and a commit.

popCommitJoin. Do a pop, a commit, and remove the first message from the message queue.

pushAtom(A). Push the atom A.

pushCatch(*Offset*). Push the address of `PC`+ *Offset* as a catch.

pushFloat(F). Makes a float of F.

pushInt_{n}.[1] ($0 \leq n \leq 15$) Push the integer n.

pushInt1(N). Push the integer N ($N < 256$).

pushInt4(N). Push the integer N.

pushIntN($N, D1, D2, ..., DN$). Push the (bignum) integer with the (16 bit) digits $D1$ to DN.

pushNil. Push the empty list nil.

pushStr($N, E1, E2, ..., EN$). Create a list of length n on the heap, containing the elements $E1$ to EN.

pushVar_{n}.[1] ($0 \leq n \leq 15$) Push the variable n on the stack.

pushVarN(N). Push variable n on the stack.

ret. Return from a function. Pop the call frame from the stack and move the return value to the new top of the stack.

save. Look at the next message in the message queue.

self. Pushes the process id on the stack.

send. The top of the stack contains a message followed by the recipient of the message. Send this message.

setTimeout. Set the timeout to be used with Wait1.

stack_need(N). Make sure that there is space for N words on the stack.

storeVar_{n}.[1] ($0 \leq n \leq 15$) Put the top of the stack in variable n.

storeVarN(N). Put the top of the stack in variable N.

tail. Take the tail of the list on the stack.

test_{integer,float,number,atom,...}. Test the type of the top of the stack.

try_me_else(*Offset*). Set FAIL_PC to PC+*Offset*, and save STOP.

try_me_else_fail. Set FAIL_PC to NULL and FAIL_REASON to "Bad match".

type($Mask$). Check that the top of the stack contains an argument of a type in the mask $Mask$.

unpkList. Get the head and the tail of the list on the stack.

unpkTuple_{n}.[1] ($0 \leq n \leq 15$) Get elements of tuple of size n.

unpkTupleN(N). Get elements of tuple of size N.

wait. Push the next message in the queue onto the stack, or reschedule it if there are no messages.

wait1(*Offset*). As wait, but if a timeout has occured set `PC` to `PC`+ *Offset*.

Appendix B

Translation scheme

This appendix contains examples of the three-address code that some JAM instructions are translated to. Since the compiler uses a virtual stack, references to the stack are turned into register references by VSP(x), which returns the register at the xth stack position.

Names inside < and > are addresses either inside the emulator or to libraries of native code. The name `emu_arg` refers to an array in the emulator that are used for passing arguments to routines in the emulator.

Word in all capital letters are either registers (*e. g.* STOP) or constants (*e. g.* INTEGER).

The function get_arg(N) returns either a register or a memory reference, depending on whether argument N resides in a register or not.

Words inside curly brackets ({,}) are to be treated as meta variables.

JAM-instructions	Three-address instructions
alloc(N)	Generates no code
argN(N)	`VSP(0) := get_arg(N)`

Table B.1: Stack instructions

JAM-instructions	Three-address instructions

comp_gt	`r1 := get_arg(1)`
	`r2 := get_arg(2)`
	`r3 := r2 & MASK`
	`if r3 != INTEGER then L5`
	`L3: r4 := r1 & MASK`
	`if r4 != INTEGER then L5`
	`L4: r1 := r1 << 0x4`
	`r2 := r2 << 0x4`
	`r1 := r1 - r2`
	`goto L6`
	`L5: save_state`
	`call <test_gt>`
	`r1 := restore_state`

	With else clause
	`L6: if r1 <= 0x0 then label1`
	`... Success`
	
	`label1:`
	`... try_me`

	Without else clause
	`L6: if r1 > 0x0 then L7`
	`... Success`
	
	`L7: EMU_ARG[0] := BAD_MATCH`
	`STOP[0] := r1`
	`STOP := STOP + 0x4`
	`goto <emu_fail>`

commit	Generates no code
dup	VSP(0) := VSP(-1)

Table B.2: Test Instructions

JAM-instructions	Three-address instructions

die	`EMU_ARG[0] := KILLED`
	`STOP[0] := VSP(-1)`
	`STOP := STOP + 0x4`
	`goto <emu_fail>`

Table B.3: Process instructions

JAM-instructions	Three-address instructions

arith_{op}	`t1 := VSP(-1) >> 0x1c`
	`if t1 != 0x1 then goto L5`
	`L1: t2 := VSP(-2) >> 0x1c`
	`if t2 != 0x1 then goto L5`
	`L2: t3 := VSP(-1) << 0x4`
	`t4 := VSP(-2) << 0x4`
	`t5 := t3 >>? 0x4`
	`t6 := t4 >>? 0x4`
	`t7 := t5 {op} t6`
	`t8 := t7 >> 0x1b`
	`if t8 != 0x0 then goto L4`
	`L3: t9 := t7 | INT_TAG`
	`goto L6`
	`L4: t7 := t7 not MASK`
	`if t8 == 0x1f then goto L3`
	`L5: save`
	`ARG_REG_1 := VSP(-1)`
	`ARG_REG_2 := VSP(-2)`
	`call <arith_{op}>`
	`t9 := restore`
	`L6: VSP(-1) := t9`

Table B.4: Arithmetic instructions

JAM-instructions	Three-address instructions

apply_call	`save`
	`STOP[0] := VSP(-1)`
	`STOP[4] := VSP(-2)`
	`STOP[8] := VSP(-3)`
	`STOP := STOP + 0xC`
	`goto <emu_apply_call>`
	`restore`

apply_enter	`STOP[0] := VSP(-1)`
	`STOP[4] := VSP(-2)`
	`STOP[8] := VSP(-3)`
	`STOP := STOP + 0xC`
	`goto <emu_apply_enter>`

call_local	`if ST_MAX >= STOP then L2`
	`L1: save`
	`call <inc stack>`
	`restore`
	`L2: REDS := REDS + 0x1`
	`if 0x7d0 > REDS then L4`
	`L3: save`
	`call <swap out>`
	`restore`
	`L4: save`
	`STOP[0] := VSP(-1)`
	`t1 := &L5`
	`STOP[4] := t1`
	`STOP[8] := 0x0`
	`STOP[12] := ARGS`
	`STOP[16] := VARS`
	`ARGS := STOP`
	`STOP := STOP + 0x14`
	`VARS := STOP`
	`call &fun`
	`L8: VSP(-1) := restore`

Table B.5: Call instructions