The Design and Implementation of a High-Performance Erlang Compiler*

Thomas Lindgren Christer Jonsson

ASTEC report 99/05
November 19, 1999

Abstract

This paper describes the design decisions and implementation of a native code compiler for Erlang, a concurrent functional language. The compiler translates byte codes into three different intermediate formats, culminating in Sparc assembly code, which is dynamically linked into the system. We critically examine our experiences with the decisions taken.

1 Introduction

Erlang is a functional language supporting concurrency, heterogenous distributed execution, and soft real-time programming [1]. It is in daily use in switches, call centers and internet servers, and similar high-availability products [2]. The Erlang implementations in use today are based on byte code emulation for portability, which hampers execution speed; a serious problem for high-performance applications.

We have investigated Erlang implementation since 1996. The compiler described in this document was developed as a second system, a "reaction" to a previous implementation by Jonsson and Johansson, Jerico [3]. Jerico was written in C and translated JAM byte codes into Sparc assembly. In the process, it performed constant propagation, dead code elimination and similar optimizations, followed by simple register allocation and dynamic linking.

The current system, Hipe (for High-Performance Erlang), grew out of this effort. Hipe was designed, built and debugged by the authors in 1997 and 1998. (Erik Johansson concurrently wrote the dynamic linker of Hipe, which is not described in this paper), and successfully ran call setup and call release for the AXD 301, an ATM switch developed in Erlang by Ericsson, during late summer 1998.

*This work was done while Lindgren was at the Computer Science Department of Uppsala University and at Ericsson Telecom AB, and Jonsson was at the Computer Science Department at Uppsala University. Lindgren is currently at Bluetail AB; Jonsson is currently at Apicula AB.

In this paper, we give an overview of the design decisions of our second system and our implementation experiences. We then discuss their successes and failures. The performance of compiled code is not treated quantitatively; this is described elsewhere [8, 7]. These papers also describe some details of the runtime system which we shall gloss over, such as exception handling and mixed native/byte code execution.

2 Overview

Much as a conventional compiler, the Hipe compiler consists of a series of phases, translating between various intermediate formats. In contrast with Jerico, which was developed in C, we decided to write the compiler in Erlang itself, to provide for faster development and greater robustness. Several design decisions were taken as reactions to problems with the Jerico implementation.

2.1 Intermediate formats

In Jerico, the intermediate format (IF) was closed to developers and it was difficult to write and debug new compiler passes. This was a great drawback once we had passed the proof-of-concept stage. To encourage experimentation, the Hipe compiler instead was designed with well-formed IFs and a flexible main loop to accomodate new passes and optimizations. Each IF was intended to catch a particular class of optimizations.

All of the IFs are based on control flow graphs (CFGs), where nodes are basic blocks and arcs represent transfers of control. In contrast with ordinary CFGs, the Hipe CFG can have multiple entry points. This is used to handle exceptions. A procedure can have several failure entry points (one per catch) in addition to the normal call entry point. If a callee throws an exception caught by the current function, control will enter the function by one of the failure entry points.[1]

Icode. The Icode IF assumes infinite registers and an implicit stack. There are few datatypes or constants; building tuples, arithmetic and similar operations are performed by function calls. Function calls take any number of arguments.

The simplicity and small size of Icode means it is suitable for initial simplifications, type analysis and type optimization, and inline expansion. These optimizations are further discussed below.

RTL, RTL2. The register transfer language RTL is a machine-independent IF similar to three-address code [4], intended to capture the conventional compiler optimizations. The actual RTL instructions are similar to the MIPS instruction set architecture [9].

[1] In retrospect, this decision made a number of compiler algorithms more complex. For example, dominators are no longer straightforward, and a number of optimizations and IFs, such as SSA form, rest on dominators.

There are tagged and untagged registers. A tagged register holds a proper Erlang value, while an untagged register holds an address, a raw integer or some similar value. Untagged registers may never live over function calls (including calls to the garbage collector), since that would make GC too complex or too inefficient. The compiler enforces proper use of tagged and untagged registers.

In RTL, the stack is implicit. RTL2 makes the stack frame and its contents visible, which can enable further optimizations and simplifies translation into Sparc code.

Sparc. The final format is an abstract Sparc assembly language. Some extra operations are added (e.g., load_atom) which are resolved by the linker.

2.2 Translation steps

On the way to native code, there are a number of translation steps.

Jam to Icode. The function is disassembled from byte code into symbolic JAM code and translated to the Icode IF. The JAM is stack based, while Icode is register based; the translation uses a virtual stack to assign registers to stack slots. The JAM also has an implicit failure continuation, which is translated by passing a fail-label to all tests. Common operations, such as element/3 or pattern matching, are inline-expanded into tests and fetches. Some obviously poor sequences of JAM code are directly translated peephole-style into more efficient Icode sequences than the concatenation of their isolated translations. Message receive operations are translated into Icode loops.

Icode to RTL. The translation of Icode into RTL means large number of operations (e.g., simple arithmetic, conses, tests) are inline expanded. Data structures are turned into loads and stores. Tagging and untagging operations become explicit. Exception handlers are expanded into code. The tagging scheme is made explicit in RTL so that constant propagation and folding can be performed.

RTL to RTL2. The RTL to RTL2 translation introduces the stack frame into the code. At a call, only the live variables are pushed on the stack; after a call, only the variables used before the next call are popped from the stack.

RTL2 to Sparc. The translation of RTL2 into the Sparc IF is straightforward. Some purity is lost in the process: the Sparc uses condition codes rather than direct tests, requires the use of continuation pointer registers, and so on.

2.3 Optimizations

Hipe performs a number of common optimizations. The following optimizations are done on Icode and the RTLs.

Unreachable code elimination. Unreachable basic blocks are deleted.

Dead code elimination. Side-effect free operations that write unused variables are eliminated.

Constant propagation and folding. Propagate and fold constants into operations.

Copy propagation. Eliminate copies $x = y$ by substituting y for x when possible. (If x becomes unused as a result, the copy operation can be deleted.)

Most of the optimizations work on extended basic blocks (EBBs, maximal trees in the CFG) rather than by fixpoint iteration, in order to save compilation time (an EBB can be analyzed and optimized in a single pass). This was probably a too conservative decision in retrospect: while outright loops are uncommon in Erlang code, pattern matching generates CFG joins, which cut off EBBs. This hinders optimization of complex pattern matching, which is common in larger applications.[2]

When we arrive at the Sparc level, only one major optimization is performed: register allocation. This is done by standard graph coloring over the entire CFG. Since the interference graph has size quadratic in the number of nodes, register allocation can be slow or even problematic for large functions.

The compiler subsequently schedules delay slots and calls the assembler and linker.

2.4 Profiling

It is straightforward to add profiling counters to the code. The dynamic linker provides directives to declare a scratch area for profiling counters, and code is easily added to increment these counters. Our profiler adds counters to all Sparc blocks and saves the Sparc CFG inside the system; after profiling is finished, the Sparc CFG can be retrieved, annotated, reoptimized and relinked. (We have currently only used the profiler to study execution characteristics of the low-level code.) Storing all CFGs in memory does not scale to large programs, where the CFG should be stored on disk; this is straightforward to add. The overhead of naive basic-block profiling was found to be about 30%.

A second, coarse-grain profiler was developed by Lindgren and Johansson. This works by adding a few builtin functions to set up the performance counters of the UltraSparc-I and -II, but was only used to measure coarse-grain events

[2] We would also like to encourage the use of complex pattern matching, so generating good code for it is important.

(e.g., the number of cache misses over an entire benchmark). This was due to the high cost of reading the performance counters, and the cost to count the number of events (this required a number of subtractions and adds, as well as extra registers and a swelling in code size).

3 Implementation

We extended Erlang with builtin functions to destructively update terms (a capability not normally found in the language). Based on these primitives, we built persistent, constant-time access vectors and hash tables. As the compiler has developed, these hash tables have become the dominant representation for tables and graphs.

Since a number of optimizations are common to all three IFs, we wanted to implement a single, common optimization pass and instantiate it with the particular IF in question. This proved difficult in Erlang; we could have written it as a callback module (as is done in OTP, for example), but that would impose an extra cost. What we wanted were *parametrized modules*, to be instantiated at compile-time and then used for free at runtime. It was simple to implement a workaround using include files – the optimizer body is copied into the instance file along with suitable macro definitions – but we feel that the solution exposes a problem with Erlang's module system.

Code sizes are shown in the Appendix.

4 Discussion

4.1 Successes

The resulting native code compiler generates efficient Sparc code. It is easily extensible: new passes can be integrated efficiently, and using Erlang means it is easy to code the optimizations once designed. Hipe compiles larger programs than Jerico, and seems not to suffer from the 'second system' syndrome.

Using three IFs turned out to be a good decision: Icode is a simple translation target; RTL can express machine-independent optimizations quite well and is easily retargeted; the Sparc format is simple and to the point.

4.2 To be revised

We found that a number of optimizations are repeated in all the IFs. These are constant propagation and folding, dead code elimination and some simplifications. We did not expect this development, but have found that it is essential.

The stack handling scheme is more important than we thought: for small programs, execution stays in the leaf functions of the call tree, but as one handles larger and larger applications, more and more time is spent in non-leaves, essentially just pushing and popping the stack between calls. Our experiments with reducing the stack frame size from four mandatory words to one were very

successful – at least as great a saving as dataflow optimizations in Jerico. Better stack handling will probably yield a considerable improvement on that.

Common subexpressions were added to RTL, and had the effect of removing the occasional superfluous untagging operation. As it turned out, the code was typically branch dominated pattern matching, so the effects on runtime were small.

4.3 To be added

Performance can sometimes hinge on inlining the right operation. More arithmetic buitins need to be inlined.

Hipe does not include a way to save compiled code, which means applications have to be entirely recompiled every time they are run. An unfortunate consequence has been that compiler optimizations have tended to be written for simplicity and speed (to reduce debugging and compile time), rather than power.

So far, experimental forms of code motion have been added. We believe, based on our experiences with these optimizations, that general code motion can be quite useful. For example, we have found some tight inner loops (simple tail recursive functions) that would benefit from code motion of constants out of the loop. Partial redundancy elimination would take care of this, if applied at the Sparc level.

Further optimizations are motivated on the Sparc level.

Global scheduling. The UltraSparc is an in-order processor, and can profit greatly from reordering instructions or moving them between basic blocks. In particular, moving loads earlier in the instruction stream provides some load latency tolerance, while reordering instructions smooths the instruction dispatch of the CPU.

Coalescing. A number of copy instructions are introduced by the translation, which are impervious to forward copy propagation, and so are best removed by coalescing during register allocation.

Basic block optimizations. Translation to Sparc introduces some constants and code that can be constant folded or eliminated as dead code. (This code is not visible on the RTL level, so similar optimizations on RTL do not help.)

Apart from conventional compiler optimizations, more work is needed on eliminating branches from code. Conditional branch elimination is fruitful in poorly-compiled pattern matching (as is the case in our current JAM frontend) and code that performs redundant type tests (which sometimes can't be eliminated by the programmer, e.g., at calls to builtin functions).

Another interesting area is low-level optimizations that take advantage of Erlang's semantics. For example, stores to the heap only initialize structure fields,

never[3] overwrite them. This can be used to improve scheduling by removing redundant memory-memory and branch-memory dependences.

Measurements of the call setup/release code of AXD301 [8] found that native compiled code spent a large number of cycles stalled waiting for instruction and data caches; given a *perfect* memory system, the system would run 71% faster. We believe that techniques that reduce instruction cache and data cache misses are vitally important. While out-of-order superscalar engines can tolerate memory latency better than the in-order UltraSparc used in our experiments, the CPU-memory latency gap is widening very quickly.

Finally, an optimizing frontend (e.g., using high-level transformations and type analysis) could drastically improve the code submitted to the compiler. While using the JAM as a frontend means source code is not needed for native compilation, compiling Erlang directly into Icode is intriguing (perhaps via Core Erlang [5]). The Hipe group is currently developing such a frontend.

4.4 False starts and dead ends

We implemented a number of experimental optimizations that did not turn out well.

Partial dead code elimination. There were many code sequences, generated by pattern matching, where (a) all elements of a structure were loaded, (b) one element of the structure was tested, and (c) if the test was false, all the work discarded. We assumed that these loads were expensive, and should be sunk below the test, so as to perform them only when necessary.

We implemented partial dead code elimination, but found that it resulted in a severe slowdown, sometimes by integer factors. The problem is that loads are sunk to their successors, even if the branch is heavily biased towards the successor where the loads are live. The only effect is then to delay useful loads, which severely slows down the common path.

A better implementation of PDCE would use execution frequency estimation to avoid moving loads into high-frequency basic blocks. Another solution would be to employ global instruction scheduling to speculate the loads sunk by PDCE, though this seems inelegant.

Type analysis. We implemented an interprocedural per-module type analyzer, in the hope of deleting unnecessary type tests. Erlang's hot code loading (where a single module can be replaced at runtime) meant we did not consider analysis of several modules at a time: a call to another module could conceivably be replaced by some function of some entirely different type later on, which would invalidate optimizations based on type.

Unfortunately, we found that most modules we tried (e.g., in the system standard libraries) were quite open to the rest of the world. This made

[3] Well, hardly ever.

the input parameters of most functions unknown, resulting in little or no type information.

We concluded that realistic type analysis either (a) spans several modules or (b) should be intraprocedural. Lindgren subsequently developed a method for merging modules while preserving hot code loading [10], which means the compiler first merges modules, then optimizes the merged clump of functions heavily.

A source-level module merger has been written by Richard Carlsson, but it has at the time of writing not been integrated in the system. Several issues about what modules to merge remain to be resolved.

A more powerful type analyzer is being developed by Sven-Olof Nyström. This analyzer potentially overcomes some of the problems of our type analysis by generating multiple versions of functions. It is an open question how to integrate such an analysis with the Hipe compiler; in particular, the AXD301 mesurements highlight the importance of the I-cache, and the interaction between I-cache and multiple versions of functions is unclear.

Non-faulting loads. We investigated non-faulting loads, as available on UltraSparc-I and UltraSparc-II. We can speculate a load over its type test if we mask out the tag and convert it to a non-faulting load. The subsequent test then decides whether the result is useful. Speculative load scheduling can be highly useful, since loads often are on the critical path.

We found that *unaligned* non-faulting loads, as would occasionally result from the technique above, cost on the order of 1000 cycles on the UltraSparcs. We conjecture that there is an exception while executing the load, which is masked by the operating system. This high cost makes load speculation an uncertain affair. (Other architectures and implementations may change this trade off.)

5 Conclusion

We have described a flexible optimizing native code compiler for Erlang. The compiler is structured around three intermediate formats: Icode, RTL and Sparc. These three tiers provide a separation of concerns that has simplified subsequent development and extensions.

A number of optimizations have been evaluated and discarded during the development of the compiler and its testing on industrial code. We have found that simple, conventional optimizations provide a large benefit. Surprisingly, some intuitively appealing optimizations (such as partial dead code elimination of loads) did instead worsen code.

We see a number of extensions: the system should be retargeted to the IA-32 architecture to extend its audience; a more powerful frontend should be provided; more low-level optimizations should be written, including machine-specific optimizations.

5.1 Acknowledgements

Erik Johansson developed the dynamic linker of Hipe and wrote a number of BIFs, including those for performance counters. He also helped with debugging the system and creditably took over the AXD measurements [6]. The current members of the Hipe group is extending and improving the Hipe system described here in several directions, some of which have not been mentioned in this paper. In particular, the efforts by Mikael Pettersson are appreciated.

Thomas Lindgren would like to thank the AXD301 team, in particular Thomas Lindquist, Ulf Wiger, Peter Lundell, Mats Cronquist and Kurt Johansson, for their help during his stay the AXD301 project.

Uppsala, November 1999

References

[1] J. Armstrong, R. Virding, C. Wikström, M. Williams. *Concurrent Functional Programming in Erlang.* Prentice-Hall, 1993. (See also `http://www.erlang.org`.)

[2] Proc. Erlang User Conference 1999, Älvsjö, 1999. (See also `http://www.erlang.org`.)

[3] E. Johansson, C. Jonsson. *Native code compilation for Erlang.* Masters thesis, Uppsala University, October 1996.

[4] S.S. Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufman, 1997.

[5] R. Carlsson, E. Johansson, S.-O. Nyström, M. Pettersson, R. Virding, T. Lindgren. *Core Erlang specification.* Work in progress.

[6] E. Johansson. *Performance Measurements and Process Optimization for Erlang.* Thesis for the degree of Licentiate of Philosophy. Uppsala University, November 1999.

[7] E. Johansson, S.-O. Nyström, M. Pettersson, K. Sagonas. *HiPE: High-Performance Erlang.* ASTEC report 99/04, October 1999.

[8] E. Johansson, S.-O. Nyström, C. Jonsson, T. Lindgren. *Evaluation of HiPE, an Erlang native code compiler.* ASTEC report 99/03, October 1999.

[9] G. Kane, J. Heinreich. *MIPS RISC architecture.* Prentice-Hall, 1992.

[10] T. Lindgren. *Module merging: aggressive optimization and code replacement in highly available systems.* UPMAIL TR 154. Uppsala University, March 1998.

A Compiler code

We counted the number of lines of code (including blank and comment lines) in the compiler.

Name	Lines of code	Description
main.erl	453	Compiler driver
jam.erl	539	JAM bytecode to data
icode.erl	1107	IF definition
icode_cfg.erl	92	IF CFG definition
icode_ebb.erl	9	EBB propagation
icode_liveness.erl	58	Liveness analysis
icode_prop.erl	497	Dataflow optimizations
translate.erl	1303	JAM to Icode
update_catches.erl	90	Minimize save/restore at catch
rtl.erl	1211	IF definition
rtl_cfg.erl	107	IF CFG definition
rtl_ebb.erl	9	EBB propagation
rtl_liveness.erl	74	Liveness analysis
rtl_prop.erl	285	Dataflow optimizations
rtl_frame.erl	637	RTL to RTL2
icode2rtl.erl	1242	Icode to RTL
finalize.erl	550	Linearize CFG
rtl2sparc.erl	224	RTL to Sparc
sparc.erl	1623	IF definition
sparc_cfg.erl	78	IF CFG definition
sparc_liveness.erl	75	Liveness analysis
sparc_op.erl	236	IF operations definition
sparc_regalloc.erl	306	Sparc interface to regalloc
regalloc.erl	854	Graph-coloring regalloc
sparc_registers.erl	213	Regalloc info
bb.erl	47	Basic blocks (library)
cfg.inc	413	CFG (param.module)
ebb.inc	214	EBB (param.module)
liveness.inc	214	Liveness analysis (param.module)