A pragmatic approach to compilation of Erlang

Erik Johansson[1], Christer Jonsson[1], Thomas Lindgren[1], Johan Bevemyr[2], Håkan Millroth[2]

[1] Computing Science Department, Uppsala University
[2] Computer Science Laboratories, Ericsson Telecom AB

We have developed a native code compiler for Erlang [2], a concurrent functional language for embedded soft real-time applications. Measurements show that our system is roughly twice as fast as the best existing Erlang compiler, BEAM [3] on SPARC machines.

Setting. We had two design constraints when writing the compiler. First, Erlang supports code replacement at runtime: running processes may replace their code and continue running. This makes interprocedural optimization awkward, since code replacement invalidates cross-module static analyses and inlining. The current compiler emphasizes intraprocedural optimizations.

Second, code space is often at a premium in an embedded setting. For this reason, we add our compiler as a minimal extension to an existing system, JAM [1], which is built on stack-oriented bytecode emulation. The compiler acts as an *accelerator* for executing JAM bytecode, much as just-in-time Java compilers accelerate the execution of JVM bytecode. The system supports efficient mixed bytecode and native code execution, so that it is possible to trade code size for execution speed.

Compiling to native code. We compile directly into SPARC machine code, since translation into C appears to offer little benefit beyond (potentially) easier retargeting. Since our work is oriented towards low-level compiler optimizations, there is no point in hiding the machine by generating C code.

Techniques and optimizations. Our native compiler translates JAM bytecodes into native code in three steps.

1. Translate stack bytecodes into register based three-address code using Ertl's technique [4]. Some arithmetic primitives are inline expanded in this phase.
2. Perform global optimizations on the code: constant propagation, constant folding, dead code elimination and unreachable code removal.
3. Finally, generate code. Translate the three-address code into SPARC code, perform register allocation (using graph coloring) and fill branch delay slots.

The compiler furthermore optimizes function calls by passing arguments in registers and reduces the 4-word stack frame of JAM into a 1-word frame.

Results. We compared our system with BEAM [3], an Erlang compiler that generates C code (or, optionally, threaded code), and JAM [1], a bytecode interpreter. JAM is the predominant implementation in use today, while BEAM is currently being introduced.

Measuring a range of ten benchmarks of varying sizes on MicroSPARC, HyperSPARC and UltraSPARC workstations we find that, depending on the machine, our system is 1.8-2.1 times faster than BEAM (using geometric means per machine) and 5.9-8.9 times faster than JAM. The benchmarks were fully native compiled for both our system and BEAM.

We find that global optimizations, delay slot filling and passing arguments in registers yield a compound 28% improvement on unoptimized code. Reducing the size of the stack frame, returning the function result in a register and inlining arithmetic primitives provides a further 30-40% improvement.

We compare compile times and compiled code size with BEAM by compiling the entire benchmark suite. We find that our compiler generates 3-5 machine instructions for each JAM byte, yielding a code expansion of 10-20 times, and that BEAM generates code 1.6 times larger than ours, while requiring ten times the compile time (mainly due to invoking the C compiler).

	Compile time (s)	Code size (KB)	Exec. time (ratio)
Our	45	196	1.0
BEAM	464	315	1.8-2.1
JAM	11	9	5.9-8.9

We refer the reader to a companion technical report [5] for more details.

References

1. Armstrong, J.L., Däcker, B.O., Virding, S.R. and Williams, M.C., Implementing a functional language for highly parallel real-time applications, in *Proc. Software Engineering for Telecommunication Switching Systems*, 1992.
2. Armstrong, J.L., Virding, S.R., Wikström, C., and Williams, M.C., *Concurrent Programming in Erlang*. Prentice Hall, second edition, 1996.
3. Hausman, B., Turbo Erlang: Approaching the Speed of C, *Implementations of Logic Programming Systems*, pp. 119–135, Kluwer Academic Publishers, 1994.
4. Ertl, M.A., A New Approach to Forth Native Code Generation, *Proc. EuroForth '92*, pp. 73-78.
5. Johansson, E., Jonsson, C., Lindgren, T., Bevemyr, J., Millroth, H., A pragmatic approach to compilation of Erlang. Technical report 136 (revised, June 1997), Computing Science Department, Uppsala University.

This article was processed using the LaTeX macro package with LLNCS style