High-Performance Erlang

Uppsala	Ericsson
Alexander Bottema	Johan Bevemyr
Erik Johansson	Håkan Millroth
Christer Jonsson	Bogdan Hausman
Richard Carlsson	
Jan Sjödin	
Thomas Lindgren	

Sponsors: Ericsson, NUTEK, TFR

Overview.

- Measurements
- Optimizations
- The future

Measurements. Our goal is to understand the performance of real Erlang programs, and to use this knowledge to guide system building efforts.

Measure quantities. (Easiest part.) Number of processes; process lifetimes; reasons for rescheduling; number of sends; size of sent messages; . . .

Measure execution time. (Harder.) Time spent in specific functions or parts of the call graph; time spent in various system tasks; time spent by category of instructions; time spent in runtime system vs. compiled code.

Measure true concurrency. (Most difficult part: work on this is just beginning.) Measure system performance under a variety of workloads, varying message types, arrival rates, . . .

Measurements: Instrumentation and logging.

- Instrument Erlang kernel to emit log
- Stopwatch instrumentation
- Post-mortem analysis of log

Measurements taken.

- Stopwatch monitoring (BEAM/Threaded).
  - Start/stop clock at each threaded instruction
  - Calibrate measurement costs and estimate real time.
  - 3x slowdown
  - Breakdown of most frequent/time-consuming instructions

  Uses UltraSPARC `%tick` register for clock cycle accuracy.

- System snapshot taken every n ms.

- Function level time profiling.

Measurements: Benchmarks.

Sim94. Military troop simulation, several scenarios. Each agent mostly sequential. (Uppsala)

Selma. Agent trading in electronic marketplace. Concurrent program. (Uppsala)

Netsim. Simulation of AXE/10 system: one AXE/10-station sending alarms to another. Concurrent program. (Ericsson)

BEAM compiler. BEAM compiler compiling Erlang programs. Sequential program. (Ericsson)

Each benchmark 20-30 Kloc.

Measurements: Concurrent execution. We measured Selma and Netsim under JAM and Sim94 under BEAM threaded.

- At a given time, few processes are ready to run (often only one).

- Run lengths between reschedules were reasonably large or large (10's to 100's of JAM instructions).

- Reasons for rescheduling were usually `receive` statements. A very small fraction of all reduction tests ever triggered a rescheduling.

- Most messages small (e.g., 56% were less than 16 words).

- Many (but not all) sends are effectively synchronous: receiver is suspended and has empty mailbox.

Measurements: Time I.

Proportion of time spent in runtime system vs. (byte) compiled code.

Program	Runtime	Compiled code
Sim94	0.20	0.80
BEAM	0.19	0.81

Runtime includes GC, scheduling, calls to C procedures, etc.

Measurements: Time II.

Breakdown of time spent in runtime system.

Program	GC	BIFs	Scheduling	Messages
Sim94	0.40	0.33	0.00	0.27
BEAM	0.35	0.29	0.01	0.35

Note: for messages, almost all the time is copying costs.

Measurements: Time III.

Breakdown of runtime system vs total time.

Program	GC	BIFs	Scheduling	Messages
Sim94	0.08	0.07	0.0	0.05
BEAM	0.06	0.05	0.0	0.06

Measurements: Conclusion.

- Erlang programs appear to spend little time in sending messages.
- The run length in concurrent programs is relatively long.
- More time spent in compiled code than runtime system.

Recommendations.

- We conclude that near-term optimization is best targeted at *improving sequential execution*.

- Runtime system operations should be exposed to compiler: inlining of primitives and BIFs.

- . . . However, optimization of concurrency becomes more interesting once the sequential system is 2-3 times faster than today.

(When the set of benchmarks is extended, these conclusions should be rechecked.)

Optimizations.

- JIT-compiler.
  - Support dynamic code replacement.
  - Which functions should be native?
- Compiler optimizations (Hipe-0.10):
  - Stack $\Rightarrow$ registers
  - Constant propagation.
  - Dead code elimination
  - Global register allocation
  - Pass arguments in machine registers
  - Inlining of selected primitives (add, xor, element, ...)

Optimizations added in Hipe-0.26.

* Merge multiple GC-checks in same function

* Slim stack frames (4 words $\Rightarrow$ 1 word)

* Function returnvalue passed in register

* Direct calls to C code

Translation from JAM.

<table>
<tr>
<td>

```
     try_me_else(L1)
     arg(0)
     getNil
     commit
     pushInt(1)
     ret
L1:  try_me_else_fail
     commit
     pushInt(0)
     ret
```

</td>
<td>

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


L2:  r1 := 0x10000000
     RETURN(r1)
```

</td>
</tr>
</table>

Uses Ertl's method (EuroFORTH'92) to eliminate stack refs

Short-circuit failure into direct jumps

Pass (some) arguments in machine registers instead of stack

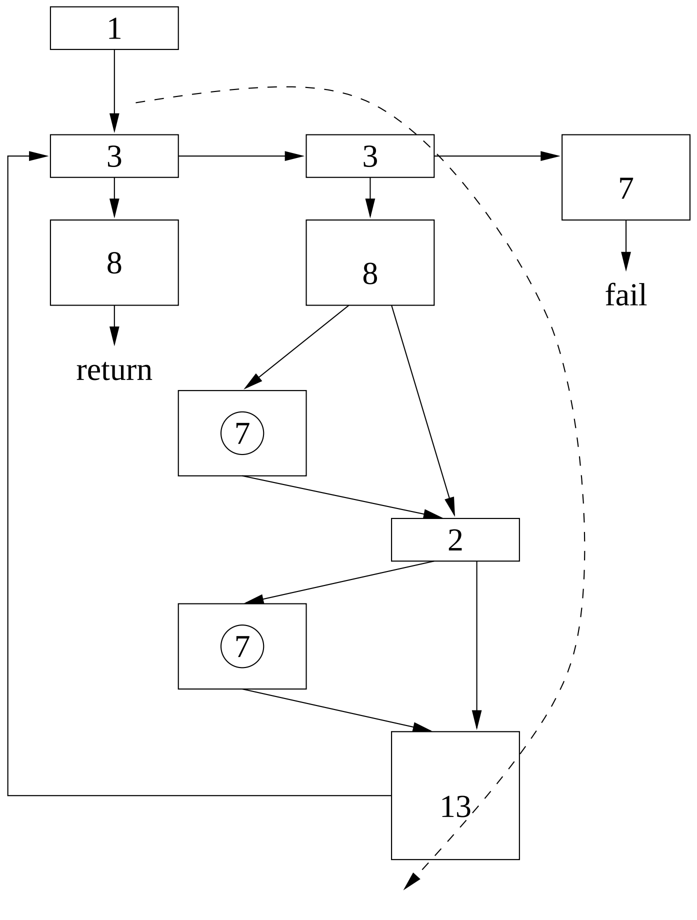
Global register allocation by graph coloring

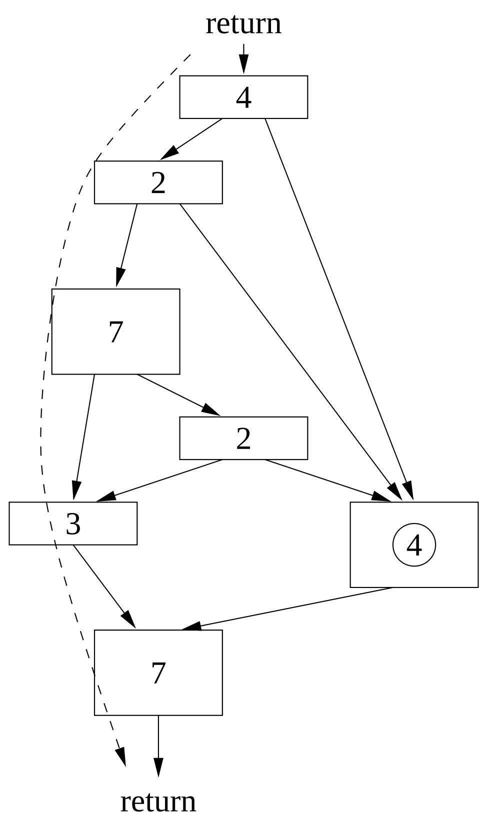
Length/1.

```
length([]) -> 0;
length([X|Xs]) -> 1 + length(Xs).
```

```
{info,foo,len,1}
   {try_me_else,1}
   {arg,0}
   getNil
   commit
   {pushInt,0}
   ret

{label,1}
   try_me_else_fail
   {alloc,1}
   {arg,0}
   unpkList
   pop
   {storeVar,{'L',{var,0}}}
   commit
   {pushInt,1}
   {pushVar,{'L',{var,0}}}
   {call,local,[],len,1}
   {arith,'+'}
   ret
```





Common path = 29 instr./iteration

Common path = 23 instr/iteration

Dataflow analyses (DFA).

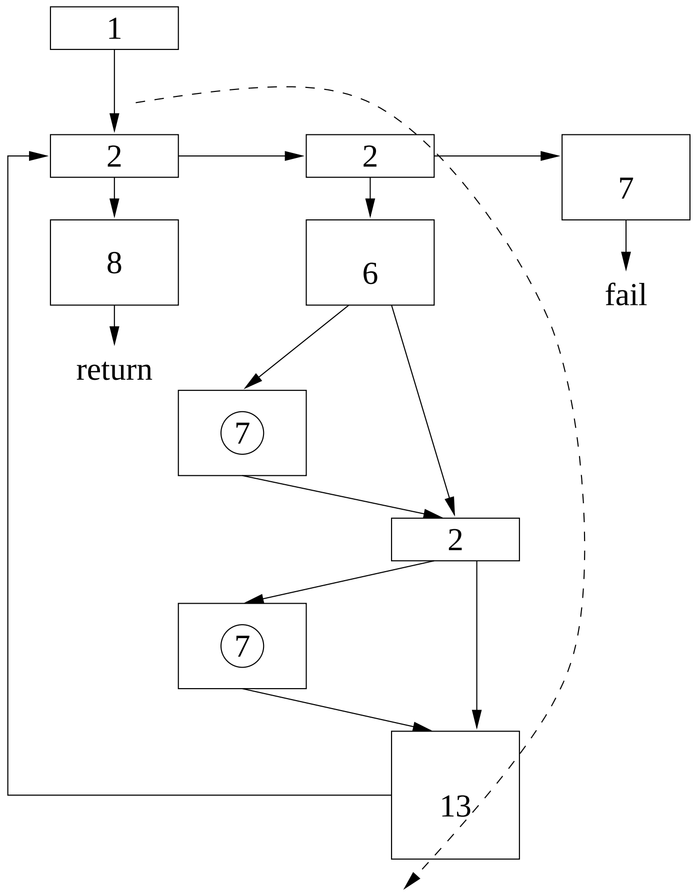
Constant propagation. Replace constant variables with their value.

Constant folding. Eliminate instructions with constant results (arithmetic + branches).

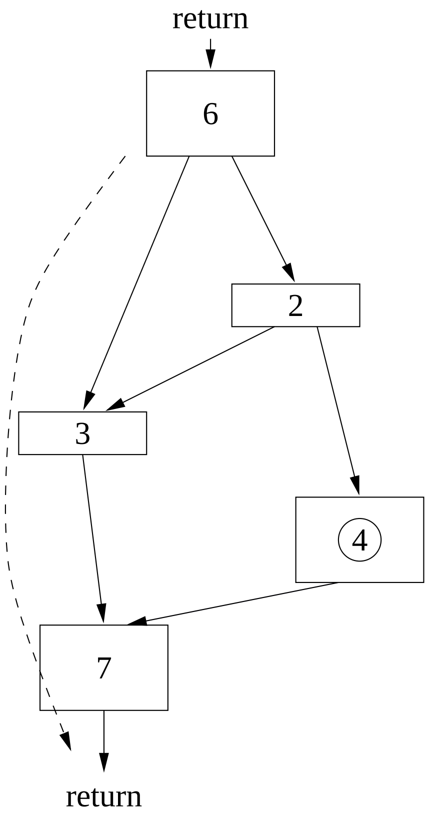
Dead code elimination. Eliminate instructions where result is never used.

Unreachable code. Remove basic blocks with no predecessors.

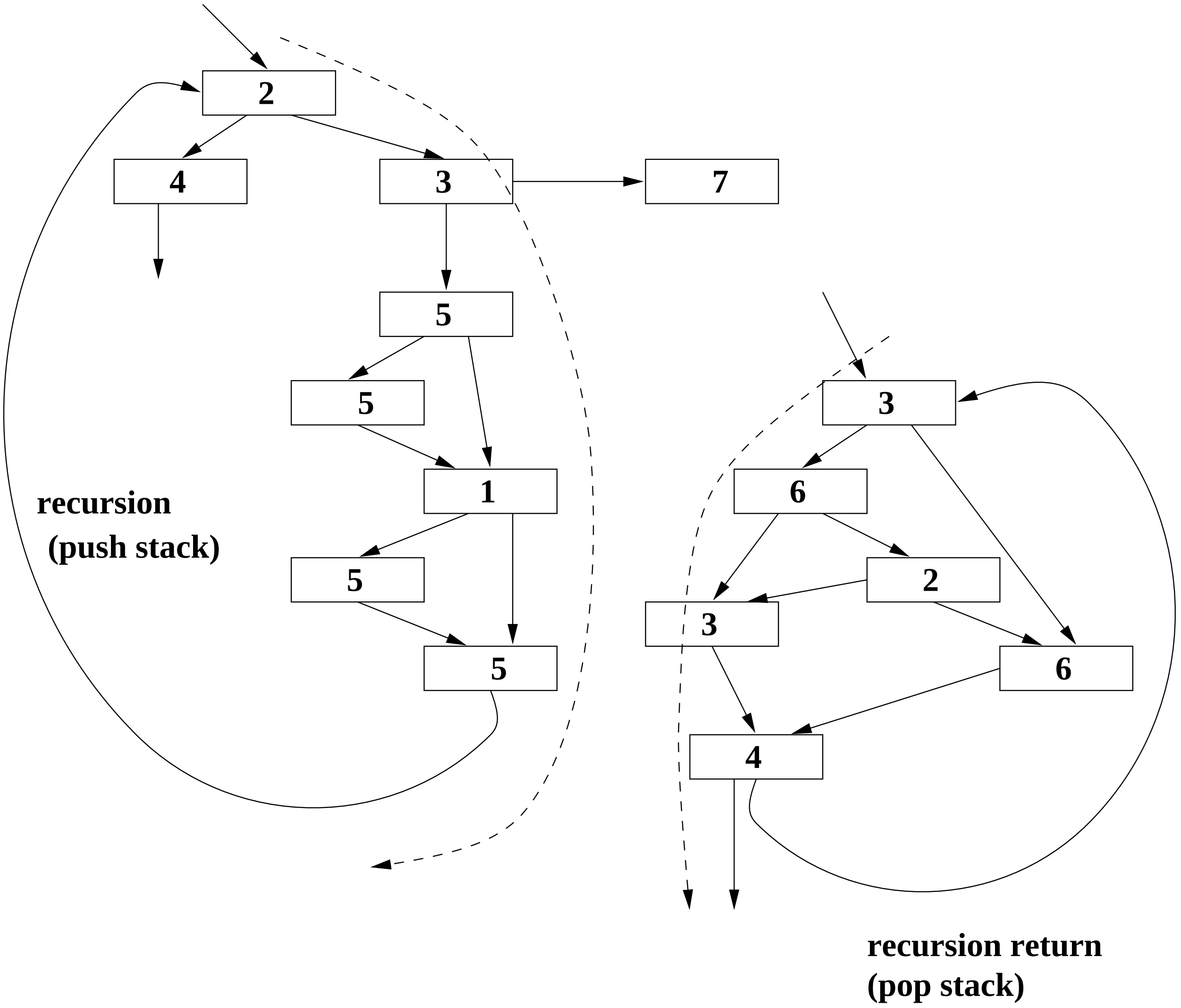
See also Aho, Sethi, Ullman (1986).



Common path = 25 instr./iteration

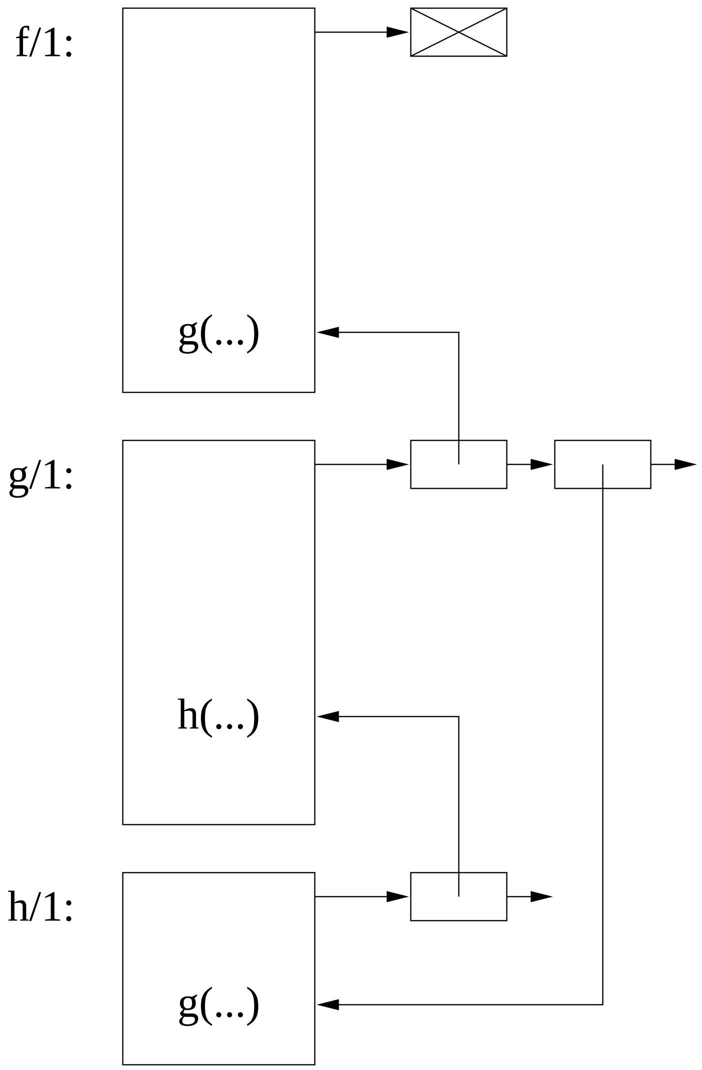


Common path = 16 instr/iteration



Common path: 16 + 16 instr / iteration

Maintaining dynamic relinking information



Effect of optimizations. Geometric mean over 10 sequential benchmarks. We used JAM 4.3.1, BEAM 4.3. Hipe-0.10 was reported at PLI-1, Hipe-0.26 at PLI-2.

Machine	JAM	BEAM	H-0.10	H-0.26
SuperSPARC	1.0	3.23	4.35	5.90
UltraSPARC	1.0	3.65	5.89	7.67
MicroSPARC	1.0	3.95	5.78	8.09

Machine	0.10/B	0.26/B	0.26/0.10
SuperSPARC	1.35	1.83	1.36
UltraSPARC	1.61	2.10	1.30
MicroSPARC	1.46	2.05	1.40

Effect of optimizations II. Counting only five larger programs: Huffman coding, Smith-Waterman string matching, raytracer, binary decoder and Barnes-Hut n-body simulation.

Machine	JAM	BEAM	H-0.10	H-0.26
SuperSPARC	1.0	2.70	3.02	4.02
UltraSPARC	1.0	2.98	3.81	5.02
MicroSPARC	1.0	3.29	3.28	4.75

Hardware	0.10/B	0.26/B	0.26/0.10
SuperSPARC	1.12	1.49	1.33
UltraSPARC	1.28	1.86	1.32
MicroSPARC	1.00	1.44	1.45

Optimizations: Summary.

- BEAM and Hipe optimize small ‘kernels’ better than larger programs (BEAM: 3.23-3.95, Hipe-0.26: 5.90-8.09.)
- Speedup still reasonable on larger programs (BEAM: 2.70-3.29, Hipe-0.26: 4.02-5.02)
- Dataflow optimizations in Hipe-0.10 yields 20-30% speedup on smaller and larger programs vs no optimization.
- Function call optimizations in Hipe-0.26 yield 30-40% speedup both on small and larger programs over Hipe-0.10.

Runtime system: Selective compilation I. BEAM compiler, JAM compiler and raytracer were used.

Program	Time (s)	Comp (ratio)	Comp (funs)
BEAMc	88.1	0	0
	70.6	0.01	3
	103.3	0.24	431
JAMc	18.7	0	0
	13.5	0.01	13
	8.3	0.22	351
tracer	33.6	0	0
	27.2	0.01	12
	18.7	0.20	73

Programs were measured on one input, recompiled, then run on another input. Percentage compiled measured vs entire system, including libraries.

Runtime system: Selective compilation II.

Program	Comp (ratio)	Time (ratio)
BEAMc	0	1.0
	0.01	1.25
	0.24	0.85
JAMc	0	1.0
	0.01	1.39
	0.22	2.25
tracer	0	1.0
	0.01	1.24
	0.20	1.80

The Future: Near future.

Traffic generation. Generate traffic for concurrent applications and measure their performance.

Optimized translation of Erlang. The translation from Erlang into three-address code will be optimized.

Dynamic compilation. Study the effects of dynamic code replacement, inlining over module boundaries, . . .

Profile-based optimizations. Run program on representative input, optimize program with knowledge of what paths/basic blocks are more likely to be executed. (Note: Freudenberger and Fisher have shown that profiles tend not to vary strongly over inputs.)

More powerful optimizer. We are developing a more powerful low-level optimizer than that currently available (Sjödin, 1997). This optimizer is currently being integrated with the Hipe environment.

Program	Estimated speedup
tracer	1.12
quicksort	1.28
fib	0.95
nreverse	1.35
smith	1.03
huff	1.64

The distant future.

- Fully understand the performance of Erlang programs.

- Optimize all aspects of Erlang for C-level performance.

- Enable “programmer friendly programming”.

estone.

BEAM (most recent)	6800
Hipe (most recent)	9200
Hipe+CSE	17400
JAM+CSE+DCE	355000

NB. The JAM execution was done on a 60 MHz SuperSPARC, the others on 144 MHz UltraSPARC.

The `estone` benchmark is vulnerable to simple, automatic high-level optimizations. We do not believe these opportunities to be commonly available in real code.

Proposal: define `ESP`, a suite of five to ten reasonably sized applications with a number of inputs to each (e.g., one short-running, one long-running). Measure total performance as an aggregate of performance per benchmark. (Cf. SPEC92, SPEC95, . . .)