OCamlCC

Raising Low-Level Bytecode to High-Level C

Michel Mauny, Benoît Vaugon
Ensta-ParisTech
Introduction

What we want to do:

- Translate OCaml bytecode into C code.

Constraint:

- Use the standard OCaml runtime.

Goals and side effects:

- Native code performances with bytecode portability.
- Post-compilation of bytecode for end users.
Compilation chain

Introduction

Generating C From Bytecode

Issues

Performances

Conclusion
Generating C From Bytecode (1)

Translation in 3 steps:

- Parsing bytecode executables.
- Performing some code transformations.
- Generating one C source file.

Decompilation:

- Translate each $\lambda$-abstraction into one C function

Optimizations:

- Whole program analysis and optimizations.
- Do not optimize code directly, generate optimizable C code.
- Static analysis based on abstract interpretation.
- Main optimizations performed:
  - Forward code pointers.
  - Remove creation of some unused closures.
  - Reduce sizes of closure environments.
  - Move values from the OCaml bytecode stack to C stack.
Generating C From Bytecode (2)

Move values from the OCaml bytecode stack to the C stack

- Transform OCaml stack cells into C local variables.
- Warning: OCaml copying GC may move memory blocks.
- A stack cell can be extracted from the stack if:
  - It is never read as a heap pointer.
  - or It is never written as a heap pointer.
  - or No garbage collection may occur during its lifetime.
- Note that some heap pointers are safely extracted from the stack.
- Effectiveness:
  - `ocamlc bootstrap`: extraction of 85% of stack cells.
Exceptions

- Use C setjmp/longjmp.
- C++ try-catch available as an option.

Tail calls

- Correct implementation of tail calls.
- GCC does not implements correctly tail calls when:
  - there is a call to setjmp in the same scope
  - or the callee receives more arguments than the caller.
- We have two implementations:
  - Architecture and GCC specific solution: assembly code.
  - Pure C solution:
    - Sub-scoping setjmp calls in local functions.
    - Using globals to pass arguments.
Issues (2)

Signal handling

- Must preserve memory consistency when calling a handler.
- Principle: polling a global flag.
- Compilation option to choose between reactivity and performance.

C compilation resources

- Generation of a single C file that \#includes the OCaml runtime.
  - Huge C file: more than $10^6$ C instructions for `ocamlc` bootstrap.
- Managable, so far.
- Separate compilation may be available as an option in future versions.
Performances (1)

The diagram shows the speed ratio of various benchmarks compiled with OCamlCC compared to other compilers. The ratio is calculated as the time taken by OCamlCC divided by the time taken by the reference compiler. A ratio less than 1 indicates better performance. The benchmarks tested include ailments, almobench, bdd, fft, ft Unsafe, fib, nucleic, ocamlc-stlib, ocamljit, quicksort, quicksort Unsafe, take, and taku.

The results indicate that OCamlCC performs competitively with other compilers, often achieving better performance, especially in benchmarks like nucleic and quicksort.
Performances

OCamlCC [Fri Sep 14 08:54:43 CEST 2012] (higher is better)

Performances (2)

![Graph showing performance ratios for different benchmarks and compiler configurations.](image)

- **ocamlcc -arch x86-64 -cc gcc**
- **ocamlcc -arch x86-64 -cc g++ -exception T**
- **ocamlcc -arch none -cc clang**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>x86-64 GCC</th>
<th>x86-64 g++ T</th>
<th>none clang</th>
</tr>
</thead>
<tbody>
<tr>
<td>almahbench</td>
<td>1.00</td>
<td>1.00</td>
<td>0.82</td>
</tr>
<tr>
<td>almahbench_unsafe</td>
<td>1.00</td>
<td>1.00</td>
<td>0.75</td>
</tr>
<tr>
<td>bdd</td>
<td>1.00</td>
<td>1.00</td>
<td>0.81</td>
</tr>
<tr>
<td>fft</td>
<td>1.00</td>
<td>1.00</td>
<td>0.97</td>
</tr>
<tr>
<td>fft_unsafe</td>
<td>1.00</td>
<td>1.00</td>
<td>0.94</td>
</tr>
<tr>
<td>fib</td>
<td>1.00</td>
<td>1.00</td>
<td>0.99</td>
</tr>
<tr>
<td>fib_unsafe</td>
<td>1.00</td>
<td>1.00</td>
<td>0.99</td>
</tr>
<tr>
<td>nucleic</td>
<td>1.00</td>
<td>1.00</td>
<td>0.96</td>
</tr>
<tr>
<td>okb</td>
<td>1.00</td>
<td>1.00</td>
<td>0.94</td>
</tr>
<tr>
<td>quicksort</td>
<td>1.00</td>
<td>1.00</td>
<td>0.97</td>
</tr>
<tr>
<td>quicksort_unsafe</td>
<td>1.00</td>
<td>1.00</td>
<td>0.94</td>
</tr>
<tr>
<td>takc</td>
<td>1.00</td>
<td>1.00</td>
<td>0.94</td>
</tr>
<tr>
<td>taku</td>
<td>1.00</td>
<td>1.00</td>
<td>0.94</td>
</tr>
</tbody>
</table>
Conclusion

Distribution

- https://github.com/ocaml-bytes/ocamlcc

Supported C compilers

- gcc (default)
- g++
- clang

As portable as the OCaml bytecode

Good performances

Future work

- Peephole optimizations:
  - Floating point arithmetics.
  - Other standard bytecode patterns.
- Other backends: icc, …