A DAIMI-Scheme VM in OCaml

Thomas Salomon and Ian Zerny
{salomon,zerny}@cs.au.dk

December 18, 2008
Revised January 2011

What have we implemented

A VM for the DAIMI-Scheme language, including

- a binary VML parser;
- a switch based interpreter;
- full support for all features and primitives; and
- dynamic compilation to x86.

The DAIMI-Scheme Language

DAIMI-Scheme is a strict subset of Scheme R5RS.

- Dynamically typed
- No floating points
- Higher order (first class functions)
- Tail call optimization
- First class continuations (call/cc)
- 54 primitives

The VML Language

VML is a byte code language
targeted by the DAIMI-Scheme compiler.

- Binary format (and s-expression)
- Register machine (tmp register plays the role of the stack)
- 10 opcodes: `nop`, `load`, `move`, `new-vec`, `extend`, `jump`, `jump-if-false`, `tail-call`, `call`, `return`

The OCaml Language

OCaml is a multi-paradigm language in the ML family.

- Algebraic data types and pattern matching
- Higher order functions
- Tail call optimization
- Efficient native code compiler
- Good foreign function interface to C

The Interpreter

- Switch based
- Approximately 100 lines of code
  (includes the parsing of opcodes and arguments)
- Slow

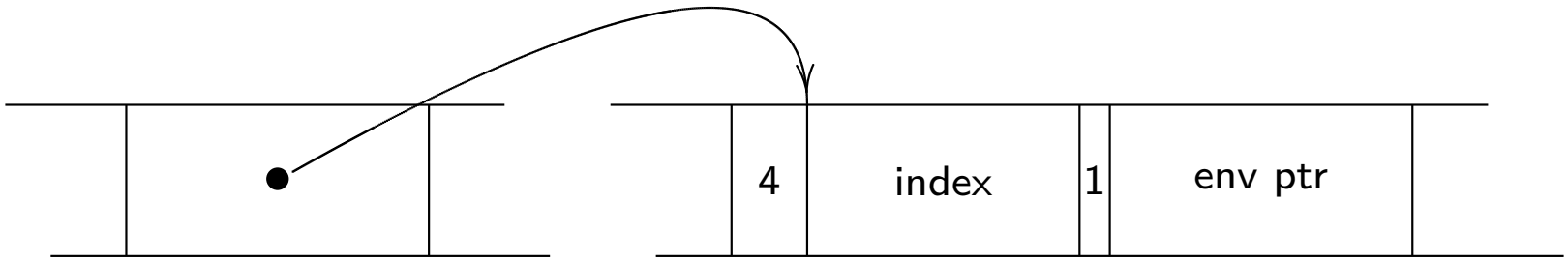
Interpreter Opcode Cases

```
match op with
| OP_NOP  → visit bits env vec cont
| OP_MOVE →
  bitmatch bits with
    { sscope : 8
    ; sindex : 16 : nativeendian
    ; tscope : 8
    ; tindex : 16 : nativeendian }
    → (get tscope env vec).(tindex) ←
        (get sscope env vec).(sindex);
      visit (dropbits (6*8) bits) env vec cont
...
```

The memory model of DAIMI-Scheme objects.

```
type value =
  | Void | Nil | True | False      (*External value types*)
  | Int  of int
  | Char of char
  | Str  of string
  | Sym  of string
  | Clo  of index * env
  | Pair of value ref * value ref (*Internal value types*)
  | Vec  of value array
  | Prim of name * arity * builtin
  | Apply | CallCC
  | Cont  of cf
and cf =                          (*Type of control frames*)
  | Init | Frame of int * env * value array * cf
and name    = string
and arity   = int
and env     = value array list
and resume  = unit → value
and builtin = value array → value array → resume → value
```

Memory Layout of a Closure Object



The Dynamic Compiler

- VML to x86 code generation (per opcode translation)
- Emitter abstraction
- Compilation strategy (per function)
- Code execution

Embedded DSL for x86

Traversing a linked list:

```
e $ mov reg_env reg_work
  (* cd...dr *)
   $ repeat n (mov (operand_disp reg_work 4) reg_work)
  (* car *)
   $ mov (operand_mem reg_work) reg_work
   $ genModify reg_work i eax
```

Syntax is created with an infix operator:

```
let ($) (x : emitter) (y : emitter → emitter) = y x
```

Interfacing with C

```
external jit_exec : string                    (* code *)
                  → int                       (* pc   *)
                  → value array               (* vec  *)
                  → env                       (* env  *)
                  → jit_result = "exec"

CAMLprim void
exec (value code, value pc, value vec, value env) {
    // Set up the special registers (S = esi, D = edi)
    asm("" : : "S"(vec), "D"(env));

    // Execute the compiled code at offset 'pc'
    goto *(&((char*)code)[Int_val(pc)]);

    // We should never reach this statement
    printf("what? executing closure jumped back...\n");
}
```

Debugging Generated Code (War Story)

Generated code for calling the mkValue callback:

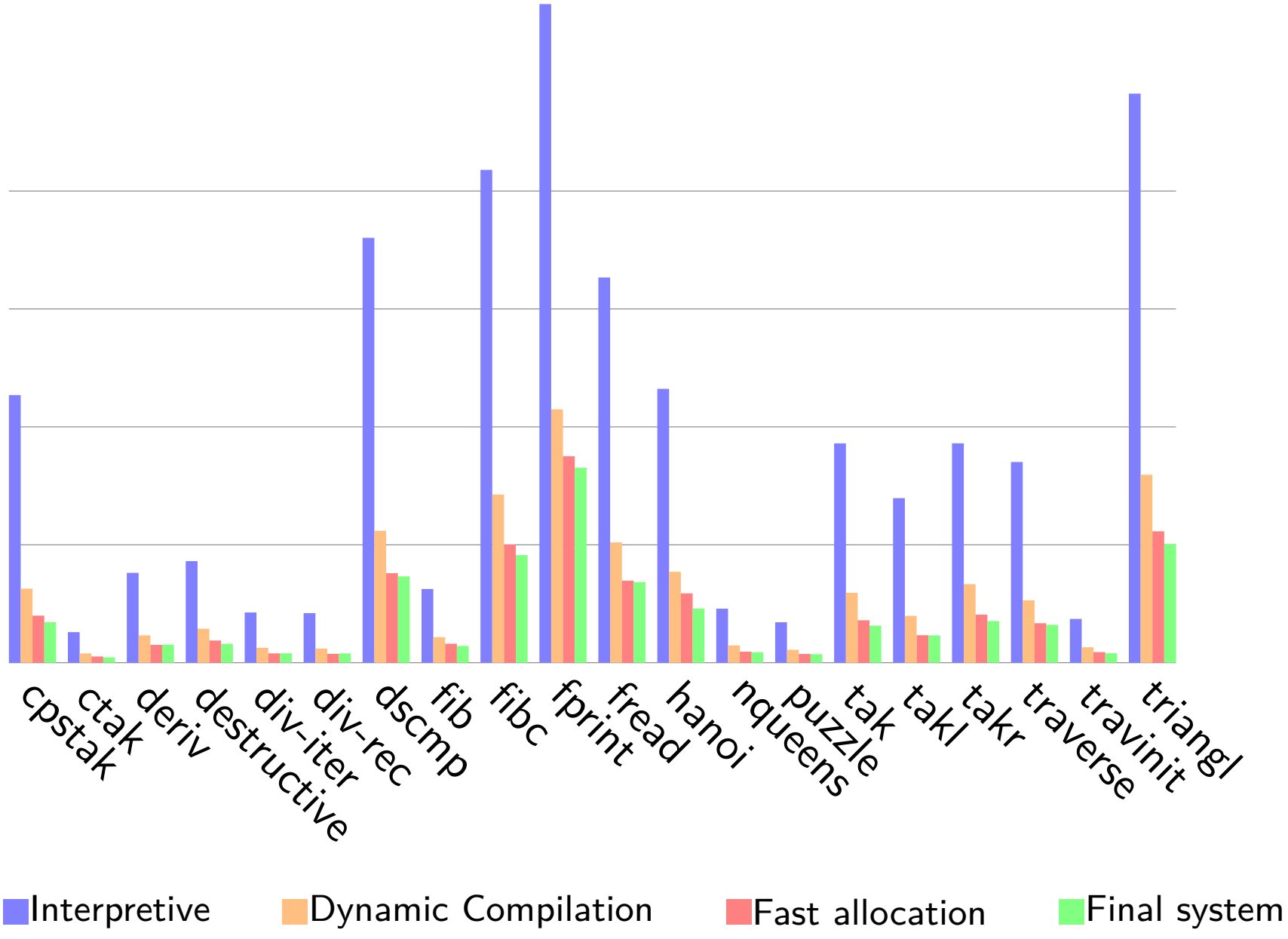
```
e $ push_pop
      [Int t; Int d; reg_env; reg_vec]
      (genCall mkValue)
```

Callback handler for mkValue in C:

```
value mkValue(int type, int data, value env, value vec) {
    CAMLparam2(env, vec);

    value result = caml_callback4(*ocaml_mkValue,
                                  Val_int(type),
                                  Val_int(data),
                                  env, vec);

    CAMLreturn(result); // the program crashes here, why?
}
```


cpstak
ctak
deriv
destructive
div-iter
div-rec
dscmp
fib
fibc
fprint
fread
hanoi
nqueens
puzzle
tak
takl
takr
traverse
travinit
triangl
Interpretive
Dynamic Compilation
Fast allocation
Final system