A DAIMI-Scheme VM and JIT compiler in OCaml

Thomas Salomon
salomon@cs.au.dk

Ian Zerny
zerny@cs.au.dk

December 2008
Revised January 2011

Abstract

The dsvmopt VM is a feature complete virtual machine for DAIMI-Scheme with a JIT compiler targeting x86. The entire VM is less than 2000 lines of code with less than 100 lines of C and it features an embedded domain-specific language for generating x86 instructions. The VM was written as part of the Virtual Machine course given by Lars Bak (Google) at Aarhus University in 2008.

Contents

List of Figures

Listings

1 Introduction

This project was created as part of the Virtual Machines course of 2008 by Lars Bak (Google) at Aarhus University. The present report documents the implementation of a virtual machine for the DAIMI-Scheme language. The virtual machine is a hybrid system with a byte-code interpreter and a dynamic compiler to x86 machine code. The virtual machine is written almost entirely in the functional programming language OCaml. Each section in this report describes a core component of the system. External resources are clarified in the appendix together with some practical information on the code infrastructure, such as building and running the system (Appendix B), tests (Appendix A.3) and benchmarks (Appendix A.4).

2 Memory

The memory model of a VM describes the representation of objects from the source language and the management of the memory they occupy. The following describes our implementation and management of DAIMI-Scheme objects. The code is found in the file `dsvmopt/values.cpp.ml`.

2.1 Object Model

The objects of the DAIMI-Scheme language (in Scheme parlor, values) are the first class objects of the language. By this we mean any value that can be syntactically created within the language or returned by a language construct. Every object must belong to exactly one of the following sets of types: booleans, integers, characters, strings, symbols, pairs, vectors, and procedures. DAIMI-Scheme is dynamically typed and all values, not variables, must have a known type at any point during the execution of a program. The most notable feature of the DAIMI-Scheme object model is that procedures are first class.

In OCaml, we have chosen to represent DAIMI-Scheme objects in terms of the algebraic data type `value`. The type definition of `value` is shown in listing 1 along with the mutually recursive type definition of control frames and a few type aliases for convenience.

```
type value =
  | Void                              (* External value types  *)
  | Nil
  | True
  | False
  | 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
  | IPort of in_channel
  | OPort of out_channel
  | EOF
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
```

Listing 1: Representation of DAIMI-Scheme Values

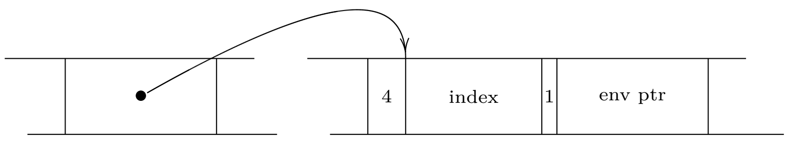

Figure 1: Memory layout of a closure object.

By using such a construct, we may reap the benefits of OCaml's high level tools for manipulating algebraic data types, such as pattern matching. From the data-type definition, the OCaml compiler generates a predictable memory layout for each of the constructors that is easily manipulated from C or assembly code. Each constant constructor (one with no argument) is represented as an unboxed OCaml integer starting from zero. OCaml provides unboxed values by tagging the least significant bit, so the garbage collector can distinguish them from pointers. In our case, we have that `True` is equal to the OCaml integer 2, and in C this is the integer 5. The non-constant constructors (those with arguments) are tagged pointers to a block of $4n$ bytes, where n is the number of constructor arguments. The tag is located in the byte preceding the block pointed to, and the constructor tags are numbered in turn, starting from 0. As such, a DAIMI-Scheme closure is represented with the `Clo` constructor and figure 1 shows how this is realized in the heap. The first cell is a pointer to a memory block of size 2×32 bits. Immediately preceding the location pointed to, is a 1-byte tag value that denotes the constructor used to create the data type. In this case it is 4, as `Clo` is the fifth non-constant constructor of **`value`**. The next 31-bits represent an unboxed integer that contains the index of the closure in our global code table. The following 1-bit is used to tag that the preceding value must not be interpreted as a pointer. The next 32-bits contain a pointer to the environment list.

Not everything sits well with this approach to representing our DAIMI-Scheme objects. An obvious problem is memory consumption. Our embedding into the algebraic data types of OCaml forces all of our objects to be multiples of 4 bytes. For some of our objects this is wasteful. In the case of characters we have at least 2 bytes that are not used for anything. Another disadvantage, is that we must be careful not to invalidate the object model of OCaml, as that would break the OCaml runtime system. This means, we cannot inject additional information in an object hash or header, without being very sure that the object remains a valid OCaml object. We do not portray to know the internals of OCaml nearly good enough to experiment with such hacks. The few problems that we have with the current memory model are far from artificial. Large objects means higher memory consumption, and the more memory we use, the more we stress the garbage collector, thereby effecting the running time of our programs. A possible solution to the above problems, is to maintain our own heap and do garbage collection ourselves. This is however outside the scope of our project and we therefore move on to other topics.

2.2 Garbage Collection

In our project we rely on the garbage collector of OCaml to do the necessary cleaning for us. Unfortunately, we found very little documentation about the current garbage collector. The most recent publication (Doligez and Gonthier, 1994) describes a concurrent garbage collector that does not seem to be the one in actual use. The most accurate description we have been able to find is by Sestoft (1994), though it too has differences with the implementation shipped with OCaml 3.10.2.

The OCaml garbage collector is a generational garbage collector with two generations. The young generation is called the *minor heap* and is used as a nursery for newly allocated objects. It is allocated once with a fixed size, and uses a *stop-and-copy* collection strategy. The old generation is named the *major heap* and may grow dynamically. The memory is handled by a linked list of memory chunks and a corresponding free-list. The major heap uses an incremental *mark-and-*

sweep collection strategy. To avoid traversing the entire major heap, in order to collect the minor heap, OCaml maintains a *reftable* that tracks all references from the major heap into the minor heap.

As noted earlier, our object model can be grouped into two categories: values of constant or non-constant constructors. Objects with a constant constructor, such as `True`, are placed as unboxed values directly into the containing structure. For objects with non-constant constructors the object is allocated on the heap and a pointer to the structure is added to the container. The internal container structures we mentioned in the above are the special "registers" described in appendix B.2. This includes a global result vector and temporary vector, along with temporary argument vectors and environment lists. All of these structures are heap allocated. The global ones, on interpreter start up, and the temporaries, as needed. We could optimize a bit here by allocating the global containers outside the heap, as they are persistent and there sizes fixed. This could reduce some work on the GC, as it would not attempt to copy the static containers around.

3 Interpreter

Before implementing the virtual machine we first evaluated various interpreter techniques in OCaml. We summarise our finding and then describe the implementation that is actually used in our virtual machine.

3.1 Exploration

Our interpreters all implement the same 5-opcode language that was used as part the Virtual Machine course. The language consists of 5 byte codes numbered from 0 to 4 that each perform some arithmetic or bit-shifting operation upon a global result value. Our interpreters can be found in the file `ocaml/interpret.ml`.

Switch-based interpreter: Straightforward switching, on the untouched byte codes, is shown in listing 2. This variant proved to be a bit faster then a comparable switch-based interpreter in C. The speed increase is most likely due to the loop calls being in tail position. Thereby avoiding the jump that occurs after a **`switch`** statement inside a **`while`**-loop in C.

```
let interp_str str =
  let rec loop i r =
    match str.[i] with
      | '0' → r
      | '1' → loop (i+1) (r+1)
      | '2' → loop (i+1) (r-1)
      | '3' → loop (i+1) (r lsl 1)
      | '4' → loop (i+1) ((r lsl 16) lor (r lsr 16))
  in loop 0 0
```

Listing 2: Switch-based interpreter.

Higher-order interpreter: An effective approach to avoid a jump, and thereby lower the amount of branch mispredictions, is to remove the jump that brings us back to the **`match`** expression. We would prefer to go directly to the case of the opcode in question. This style of implementation is referred to as *threaded code* (Bell, 1973). Such an implementation is shown in listing 3, using higher-order functions. Here the next opcode is invoked as a tail call to a member of the `ops` array, essentially implementing a direct jump to the next opcode. We list both the translation from a byte code string to an array of higher order functions that implement the byte code operations, and the "interpreter" that simply invokes the first opcode.

```
let convert_direct str =
  (*  type oplabel = int →  int →  oplabel array →  int *)
  let oplabel0 = (fun r pc ops →  r) in
  let oplabel1 = (fun r pc ops →  ops.(pc) (r+1)                        (pc+1) ops) in
  let oplabel2 = (fun r pc ops →  ops.(pc) (r-1)                        (pc+1) ops) in
  let oplabel3 = (fun r pc ops →  ops.(pc) (r lsl 1)                    (pc+1) ops) in
  let oplabel4 = (fun r pc ops →  ops.(pc) ((r lsl 16) lor (r lsr 16)) (pc+1) ops) in
  let ops = Array.make (String.length str) oplabel0 in
    for i = 0 to String.length str - 1 do
      ops.(i) ← match (code str.[i] - code ’0’) with
        | 0 → oplabel0
        | 1 → oplabel1
        | 2 → oplabel2
        | 3 → oplabel3
        | 4 → oplabel4
    done;
    ops

let interp_direct ops =
  ops.(0) 0 1 ops
```

Listing 3: Higher-order threaded code interpreter.

The use of tail calls should ideally translate to direct jumps. This is however not the case in OCaml. The assembly output of OCaml contains a, in our case, unnecessary call to the OCaml subroutine `caml_apply3`. This is however easily repaired by post-processing the assembly output of OCaml, as shown in listing 4.

```
op1:    movl    -2(%ecx, %ebx, 2), %edx  # ops.(pc)
        addl    $2, %ebx                 # pc + 1
        addl    $2, %eax                 # r  + 1
      # jmp     caml_apply3              # unwanted jump, replaced by the next 2 commands
        movl    8(%edx), %esi            # function pointer of the OCaml closure
        jmp     *%esi                    # tail call
```

Listing 4: Repaired tail calls in direct threaded code.

After performing this repair, the above code achieves a running time very close to that of a direct threaded variant in C. Some additional overhead is caused by operations for handling the tag bit on OCaml integers. This appears in the opcodes that use shift operations.

First-order interpreter: A third alternative, for an interpreter in OCaml, is to denote the byte codes in the form of a variant type. Such an interpreter is in listing 5. We have omitted the conversion from a byte code string to the opcode array for brevity. This interpreter is essentially the defunctionalized (Reynolds, 1998) variant of the direct threaded interpreter, i.e., a first order interpreter. It is also identical to the first switch-based interpreter, except for the type dispatched over. This interpreter is however faster, as the OCaml compiler knows the exact members of the type and may generate the switch code more efficiently.

```
type opcode = Op0 | Op1 | Op2 | Op3 | Op4
let interp_defun ops =
  let rec apply r pc =
    match ops.(pc) with
      | Op0 → r
      | Op1 → apply (r+1) (pc+1)
      | Op2 → apply (r-1) (pc+1)
      | Op3 → apply (r lsl 1) (pc+1)
      | Op4 → apply ((r lsl 16) lor (r lsr 16)) (pc+1)
  in apply 0 0
```

Listing 5: First-order interpreter.

3.2 Implementation

Our development method for this project has been to first make it work, and then to make it faster. Our interpreter is therefore implemented in straightforward fashion with emphasis on correctness and clarity. The VML parser, found in the file `dsvmopt/vml.cpp.ml`, parses the byte code files and hands these structures to the interpreter in the form of a few tables and a byte-string that contains the actual byte codes to execute. The interpreter code is in the file `dsvmopt/interpretBits.cpp.ml`. The `interpret` function sets up the "global" structures and proceeds to interpret the byte codes. The interpreter itself consists of the three mutually recursive functions `visit`, `application` and `continue` that are described below.

3.2.1 Visit

Most of the interpretive work takes place in `visit`. To parse the values from the byte code we use the OCaml *bitstring* library which allows pattern matching within binary data (Jones). The `visit` function parses the next opcode and switches to the case that handles that opcode. The code for handling a VML *move* opcode is shown in listing 6. Here `bitmatch` is used to parse the arguments of the *move* instruction. After parsing the arguments, we write the source value into the target cell. Finally having removed 6 bytes, corresponding to the size of the *move* opcode, we continue with a tail call to `visit`.

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

Listing 6: Interpreter code to handle a *move* instruction.

The reader might have noticed the seemingly unnecessary **`let`**-binding around the `bitmatch` expression. This binding is required since the syntactic transformation performed by the bitstring preprocessor does not respect tail calls. If the call to `visit` is not in tail position our interpreter will soon blow its call stack. It would be nice to repair the *bitstring* library to avoid this, but that is outside the scope of this project.

3.2.2 Application

The `application` function handles all applications of applicable objects. An applicable object can be a builtin function (also referred to as a primitive), a closure, or a continuation. The two builtin functions *apply* and *call/cc*, represented as the **`value`** constructors `Apply` and `CallCC` respectively, are treated specially.

- The *apply* function is used to apply a function to a list of arguments. For example, the program `(apply + '(1 2 3))` would have the value 6. In the case of `Apply` we need to convert the second argument from a first-class list, to an internal OCaml vector.
- The *call/cc* function captures the current continuation and hands it as an argument to the function supplied to *call/cc*. In the interpreter, this means coping the current control stack, denoted by `cont`, into a first-class value with the constructor `Cont`. The argument to *call/cc* is then applied to the continuation. Now, `Cont` is itself an applicable value, denoting a continuation, and when applied, it will write its argument in the result vector and return to the last frame in the control stack it is holding, thereby discarding the current control stack.

3.2.3 Continue

The `continue` function simply continues execution in some control frame. If the frame is the initial frame, it stops execution by returning the top most value in the result vector.

3.2.4 Builtin Functions

All of the builtin functions are implemented in the file `dsvmopt/builtins.cpp.ml`. Each function has the type `builtin`, as shown in listing 1, taking an argument array, the result array and a resume continuation. Listing 7 shows the implementation of the DAIMI-Scheme `integer?` primitive.

```
let integer_p actuals result resume =
  result.(0) ← mkBool
    (match actuals.(0) with
       | Int _ → true
       | _     → false);
  resume ()
```

Listing 7: Implementation of the `integer?` primitive.

3.2.5 Considerations

One problem with the current implementation is the interdependency between the interpreter and the type of values. We see that the type of control frames (i.e., `cf`) is dependent on the implementation strategy of the interpreter, but this type is depended upon by the type of objects (i.e., **`value`**). This makes it hard to implement an alternative interpreter with a different type of control frames. Possible solutions are to either add the type of control frames as a type parameter to **`value`**; or alternatively parameterise the `Values` module by the type of control frames, i.e. make it a *functor*, in ML terminology. After such a modification we could supply several interpreter implementations that could be selected by a flag to the VM. We have done neither, as our remaining time has been spent on dynamic compilation.

4 Dynamic Compilation

Our DAIMI-Scheme VM supports just-in-time compilation as its major optimisation feature. This technique involves compilation of VML byte codes into x86 machine instructions, at runtime, using closures as the basic unit of compilation. The pipeline constitutes a single linear pass over the byte codes with a fairly naive translation into target instructions. No control flow analysis or subsequent optimisations are carried out as part of the compilation.

4.1 Architecture

The dynamic compilation system is comprised of several smaller components.

Code Generation A code generator is found in the file `dsvmopt/codegen.cpp.ml`. The code generator resembles the interpreter loop in structure, and handles a linear sequence of individual byte codes and the corresponding emission of the target instructions. The actual emission is done through a code emitter object which is passed around in the code generator. The emitter object is a tuple consisting of a code buffer, which is a mutable string, a program counter indexing the buffer, a delay list to carry out instruction emission that has been paused, and a mark list to allow tagging specific addresses in the emitted code.

The code emitter module consists of several functions that lift the generation of x86 instructions to a slightly higher level of abstraction. These functions can be seen as a small domain specific assembly language which hides the somewhat complex details of x86 opcodes and arguments.

Listing 8 shows an example of a lookup, into the environment, written in the assembly language. The first line is an example of a simple machine instruction abstraction. In this case, a mov-instruction is generated which copies the contents of the register containing the environment address into a working register. The mov-instruction can take registers, integer constants, and memory operands as arguments. For instance, in the third line the result of `(operand_mem reg_work)` is given as the first argument. This line results in copying the contents of the memory location, referenced by the work register, into the work register. Similar functions exist for referencing memory addresses using memory displacement constants and scale-index-base values. The mov-function transparently handles all the specific details of the opcode layout based on the input parameters.

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

Listing 8: Traversal of a linked list in embedded assembly language.

A nice syntax for writing our assembly code can be obtained with the simple use of an infix operator. The `$`-operator is a left-associative infix operator which applies its right argument (expected to be a function) to its left. The assembly process is bootstrapped with an emitter object in the first line. As long as every subsequent function has the type `emitter → emitter` the emitter object is simply passed along the chain of assembly instructions and updated along the way. This structure also enables us to define small and simple helper functions and use them in the instruction pipeline. The function `repeat` in the second line of Listing 8 is an example of such a function. It emits the given instruction n times. The `repeat` function is curried and has type `int → (emitter → emitter) → emitter → emitter`. The result of supplying the arguments `n` and `mov (...)` is a function of type `emitter → emitter` which is of the desired type.

To support compilation of branches, we need to relate byte code addresses with the x86 machine code addresses. Adresses in VML are given relative to the beginning of the program and for simplicity every emitted x86 address is an absolute 32-bit memory location. Every time we start emitting code for a given byte code we initially call the `mark` function and pass the total size of the current byte code, wrapped in the `Opcode` data constructor, as argument. This function will extend the mark list of the emitter with a tuple containing the argument and the current pc of the emitter. Whenever we need to do an x86 branch instruction we emit the instruction with an arbitrary 32-bit address, immediately followed by a call to the `mark` function. Only this time, we supply the byte code address of the jump wrapped in the `Jump` data constructor. Listing 9 shows an example of this. When all opcodes have been emitted we can start injecting the correct branch addresses by doing a traversal of the reversed mark list.

```
| { OP_JUMP : 8
  ; offset : 32 : nativeendian
  ; bits   : -1 : bitstring }
  →
    e $ mark (Opcode 5)
      $ jmp (Int 0)
      $ mark (Jump (Int32.to_int offset))
      $ genOpcodes bits
```

Listing 9: Using the `mark` function for branching.

Compilation Strategy We have chosen to subdivide the source program into units associated with their respective functions. As such, functions comprise the basic unit of compilation. This means that the byte codes in the main program will never be compiled. However, this code typically has linear execution as loops are implemented by means of tail calls. In other words, the main program is only executed once and compiling it would most likely be a wast of time and space. This of course depends on the overall interpreter speed measured against the time it takes to compile the byte codes. As of now, we compile functions when creating a closure. The compiled code is then saved for reuse if a closure of the same function is created again. Compiling on all closure creations pays off because our interpreter is fairly slow. If this were not the case it becomes interesting to consider other more advanced strategies, such as invocation counting or sample based compilation by using a timer. Right now we have taken the simple approach, although avoiding the compilation overhead for one-shot functions would be desirable.

Code Execution Once the code has been emitted into an OCaml string, we need to be able to transfer control to it. This is done by means of the OCaml foreign function interface that enables us to call C functions from within OCaml. For this purpose we have a single C file `dsvmopt/jit.c` with a small number of functions for handling the execution of compiled code.

- An initialisation function (`init`) is called one time by the interpreter to set up the execution environment.
- An execution function (`exec`) takes the code string and an offset into the code as arguments and transfers control to this memory location with a goto statement.
- Other functions for handling instantiation and modification of OCaml values as well as handling function calls back into OCaml code.

In OCaml code the application function of the interpreter, described in section 3.2.2, is extended with a match for a compiled closure with the **`value`** constructor `Comp`. If the arity of the closure matches the argument list the `exec` function, which is yet another extension to the interpreter, is called. The `exec` function simply invokes the external C function for executing the machine code. Depending on the return value of this execution the interpreter will do one of the following actions.

- Continue with the current continuation. This happens if the closure simply reached its return statement, thus returning normally. Marked by returning a pc of 0.
- If the execution returns a closure and a `vec` array then this closure should be applied to the vector with the current continuation. This corresponds to a tail call being made from inside the generated code. Marked by returning a pc of 1.
- If the execution returns a pc larger than 1, along with a closure, `vec`, `env` and a size, we have met a call opcode. This means saving the callers execution state as a special control frame (`CFrame`), and then applying the closure to perform the call.

4.2 Working with the OCaml GC

When operating directly on the internal representations of OCaml data structures it is extremely important to know which actions can potentially trigger GC operations, in order to handle the side effects of such an operation correctly. We must also ensure conformance with OCaml's technique for differentiating pointers from non-pointers. We identify the following ways of triggering a GC call, and potentially invalidating object locations:

- memory allocation;
- callbacks to OCaml, as control is transfered to the OCaml runtime system.

Meanwhile the following invariants are required to hold at any given time while running inside the generated machine code:

- registers **`esi`** and **`edi`** (aliased as `reg_vec` and `reg_env`) must point to the current argument vector and the environment list, respectively;
- the `res`, `tmp`, `glo`, and `lib` members of the C structure, named statics, points to their corresponding OCaml data objects;

- the `current_code` variable points to the address of the code object currently being executed.

We rely heavily on the OCaml C macros in order to ensure these invariants. The `CAMLparamN` macros ensure that references are updated if the referenced objects are relocated during a GC call. The **`esi`** and **`edi`** registers are then updated by passing them as arguments to the C function and having the C function use `CAMLparamN` to register them with the GC. On return the correct references are then popped back into the registers. Global variables are easily registered with the GC by means of the `caml_register_global_root` macro. This macro need only be called one time in the initialisation phase. As code objects are OCaml data structures they too can be relocated upon GC calls. If this happens while executing a C function we get undefined behaviour when returning as the code we are trying to return to is no longer there. We therefore need to make sure that the `eip` register is restored correctly. This is achieved with the `current_code` variable which is also bound by the `caml_register_global_root` macro at initialisation and at every call to `exec` we update the `current_code` pointer. We do not need to worry about nested calls in compiled code as function calls always return fully from compiled code and function application is handled in our runtime system. To secure the return address each callback in C must locate the return address in the current stack frame and subtract `current_code` from it. This gives us a relative offset into the code block. Just before we return, we add `current_code` to the offset (`current_code` could have been changed by the GC). The resulting value is written back as the return address in the stack frame. This ensures correct restoration of the `eip` register on function return. The macros `SAVE_EIP` and `RESTORE_EIP` handle the described operations.

4.3 Optimisations

By using dynamic compilation we obtain a decrease in overall running time. This is mainly because we compact a series of byte codes, which otherwise requires a switch in the interpreter for each and every byte code, into a single entity that can be executed without the overhead of the interpreter switch. At this point we do not exploit much of the information available at compile time that could enable further optimisations.

We do however support a faster allocation mechanism based on the abstract data type being allocated. If an abstract data type is known to have a constant constructor at compile time we simply generate code for allocating the value corresponding to the data type.

4.4 Future optimisations

Exploiting the compile time information gives rise to a number of possible optimisations that would be interesting to implement.

An example of such an optimisation would be to inline builtin functions, thus avoiding an escape into the OCaml runtime system. When compiling the call instruction we can easily determine whether it is a call to a builtin function and inline the call instead. Some builtins that work on an arbitrary number of parameters, such as the arithmetic operators could furthermore be specialised for the many cases where only two arguments are given. They could even be specialised to work on unboxed integers, hereby saving a lot of memory operations. However, this is not as simple as it first seems. Even builtin functions can be overwritten in Scheme. So a protection mechanism would be needed to fallback if our assumptions did not hold.

Another interesting optimisation would be to do inline caching on closure calls as the environment is known at compile time. By doing this it is possible to specialise the closures to work on a specific environment. The generated code would need to guard the applications in case the environment has been destructively modified. In most cases however, they remain fixed, so one could expect a considerable performance gain. Also, implementing such an optimisation would require us to do separate compilation for each closure creation, instead of our current strategy of reusing compiled function code, thus maintaining several compiled instances of the same function.

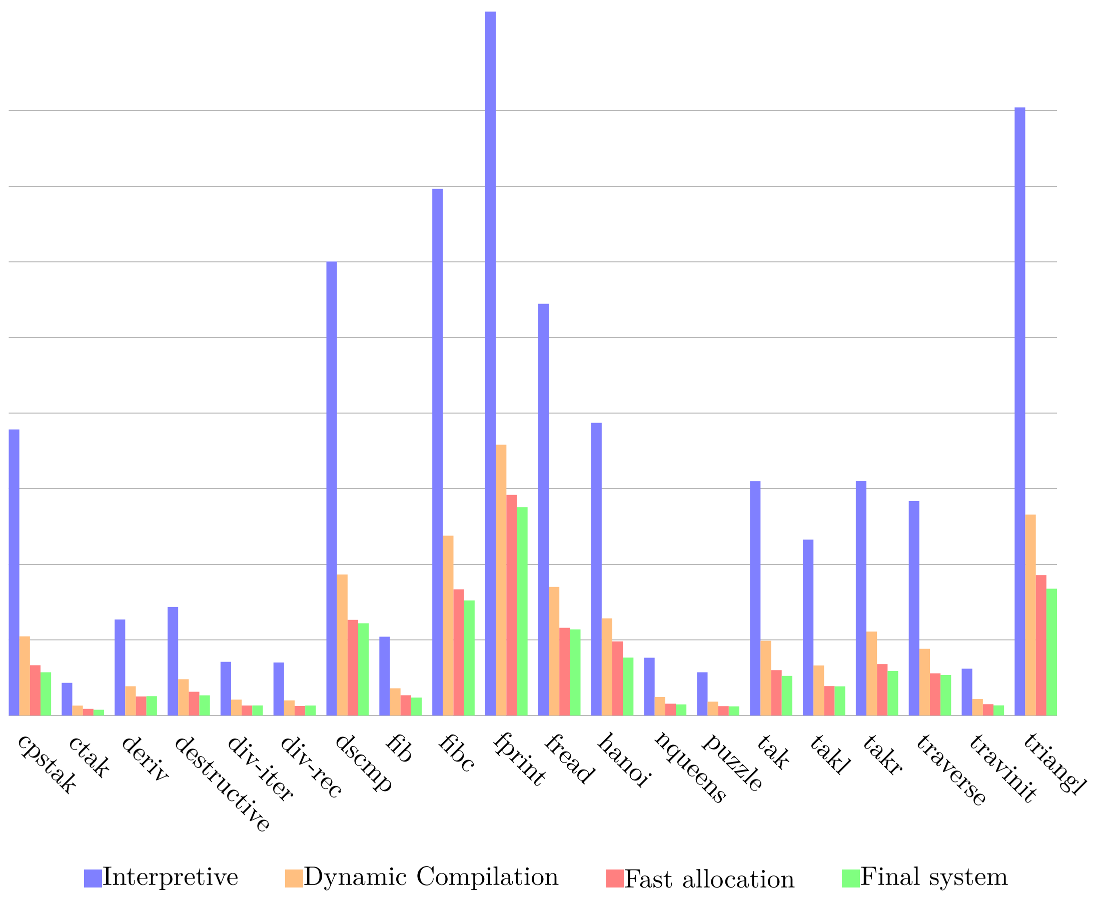

Figure 2: Benchmarks for several variants of the VM.

5 Results

During development we have used a benchmark suite to obtain some empirical evidence that our optimisations were in fact optimisations. All the benchmarks have been done on a system running the 32-bit version of the Ubuntu GNU/Linux distribution version 8.10. The system had an AMD Athlon 64 FX-55 Processor with 2613.339 MHz and a 1024 KB cache, and 2 GB of main memory. All benchmarks have been run from a terminal after shutting down the X11 graphical environment and other unwanted tasks in an attempt to minimize scheduling.

5.1 Execution Time

Some of the benchmarks are shown in Figure 2. We have included four runs showing different phases in our VM development. The first (blue) shows a purely interpretive execution and the other three with dynamic compilation enabled. The second (orange) is taken at the time the VM gained full dynamic compilation support. The last two are further optimizations. The third benchmark (red) is after implementing fast allocation, by avoiding callbacks for unboxed objects and using better allocation mechanisms for small objects and further avoiding expensive modification operations where possible. The last benchmark shows the final running time of the system after a few other optimizations, such as adding a `RELEASE` flag to avoid overhead from debugging and development code.

We have purposely omitted the *lattice* benchmark from the diagram as it is remarkably longer than the rest. All of the benchmarking data is located as text files in the `dsvmopt/marks` directory.

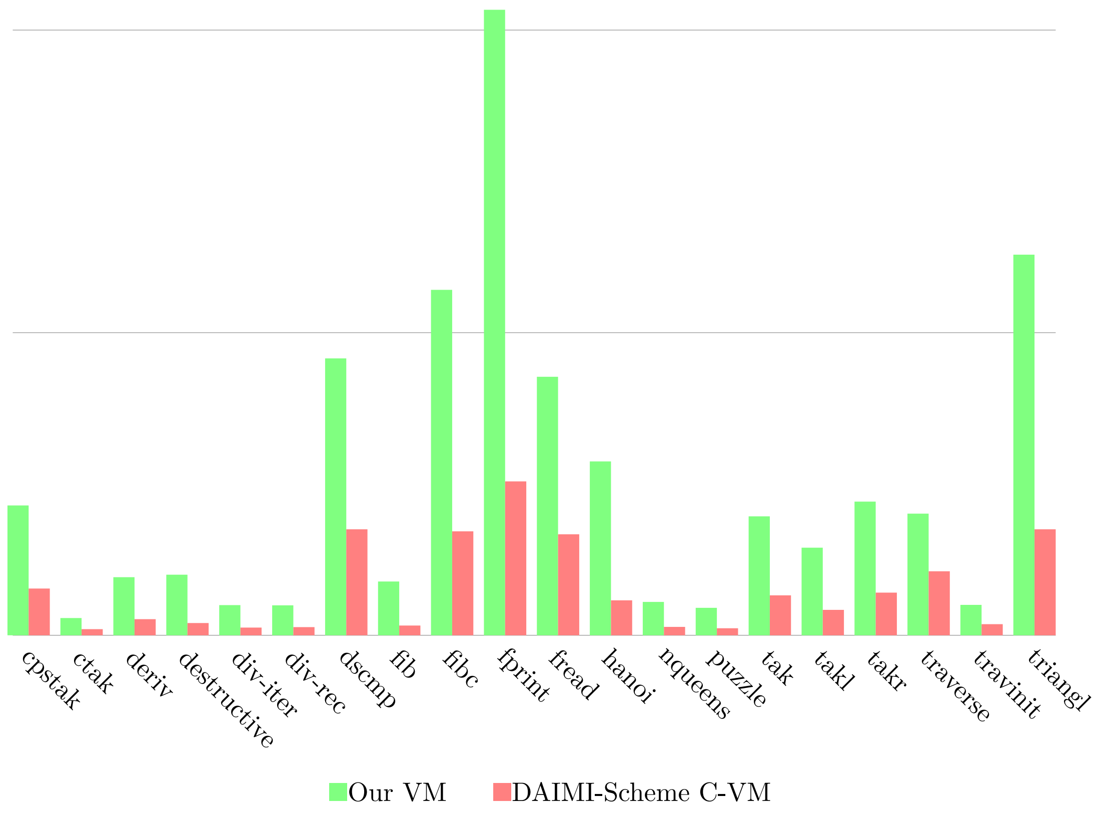

Figure 3: The DAIMI-Scheme C-VM compared to our VM.

As can be seen from Figure 2, we have achieved a significant speed up relative to the speed of our interpreter. The final result is about a factor 3 faster, and there is still plenty of room for improvement within the system. Compared to the C implementation, developed as part of Danvy's compiler course, our VM is about a factor 4 slower as shown in Figure 3. That is however less than we had expected as our first variant of the VM was much slower than the C version. Unfortunately, we only have access to the binary and can not compare implementation strategies.

5.2 Memory Consumption

Since we are using OCaml for all of our memory we have little insight into exactly where our memory is going. We have added a `-stats` flag that prints out information obtained by the OCaml GC. The memory consumption of running our VM with dynamic compilation disabled and enabled is shown in Figure 4. These numbers indicate a few puzzling things. First we can see that the total amount of allocated kilobytes falls with about 60.000.000. The only reasonable explanation seems to be that our interpreter loop (in `visit`) is allocating a lot of temporary structures. This is further supported by observing that our peek memory usage without dynamic compilation is lower than with it enabled, also the major heap is smaller. As noted in section 3.2.1, we wrap all `bitmatch` expressions with a let. This could contribute to a large amount of temporary memory usage. We have not looked closely at the syntactic transformations performed by the *bitstring* library and that too could contain some unnecessary allocations. These observations motivate a closer look at our interpreter and show that we still have much room for improvement.

	Without Jit	With Jit
Minor heap size	128 k	128 k
Major heap size	12.720 k	14.160 k
Major live memory	8.666 k	11.152 k
Max live memory	16.320 k	17.904 k
Minor allocations	65.421.794 k	5.569.045 k
Major allocations	182.377 k	50.857 k
Total allocations	65.422.535 k	5.570.788 k
Minor collections	511.111	43.513
Major collections	154	40
Compactions	5	1

Figure 4: Memory consumption for `dsvmopt/benchmarks/dscmp`.

5.3 Conclusion

The final outcome of our project is a complete virtual machine for the DAIMI-Scheme language. By complete we mean that it supports the entire language specification with reasonable performance. The VM contains one major optimization technique, namely dynamic compilation, and we have shown that it greatly improves the overall performance of the system. The entire system is written in less than 2000 lines of code, according to `sloccount`, with less than 100 lines of C. We find this remarkably low and attribute it to the expressiveness of OCaml. We have also encountered shortcomings, such as the descriptive types in OCaml. For our purpose we would have preferred prescriptive types allowing more control over the exact representation in memory.

During the project we have been exposed to two very different levels of abstraction. The high level functional paradigm of OCaml and the very low level architecture that is IA32. This has especially shown itself with respect to debugging that turns out to be extremely difficult in the latter, whereas in OCaml, programs have a tendency to work if they type check.

Our VM implementation is fairly straightforward. At several points we have noted possibilities for optimizations that we have not pursued. This has been necessary to stay on track with the goals of the project. There are considerably many ways to optimize the execution of DAIMI-Scheme programs and during this project we have developed only one.

We would like to thank Olivier Danvy for his kind help in extending the DAIMI-Scheme compiler and providing helpful comments during the project.

References

James R. Bell. Threaded code. *Commun. ACM*, 16(6):370–372, 1973.

Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 70–83, New York, NY, USA, 1994. ACM. ISBN 0-89791-636-0.

Richard Jones. The ocaml bitstring library. http://code.google.com/p/bitstring.

J. Rees (eds.) R. Kelsey, W. Clinger. Revised5 report on the algorithmic language scheme, higher-order and symbolic computation, August 1998.

John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Higher-Order and Symbolic Computation 11(4), 363–397. Reprinted from the proceedings of the 25th ACM National Conference*, 1998.

Peter Sestoft. The garbage collector used in caml light. Internet, October 1994. URL `http://pauillac.inria.fr/~doligez/caml-guts/Sestoft94.txt`.

A Usage

This appendix briefly describes how the program source is structured, how to build the programs, and how to use the programs. The source distribution has the following hierarchy

daimi-scheme contains the entire DAIMI-Scheme system. Documentation is located at the top level in several text files and the `bin` directory contains scripts to run the compiler, interpreter and assembler.

dsvmopt contains all the VM code. More precisely the VM code consists of the OCaml files: `builtins.cpp.ml`, `codegen.cpp.ml`, `emitter.cpp.ml`, `interpretBits.cpp.ml`, `main.cpp.ml`, `utils.cpp.ml`, `values.cpp.ml`, `vml.cpp.ml`;
and the C files: `defines.h` and `jit.c`.

dsvmopt/tests contains a few of our own tests, and the `runtests` script to run all DAIMI-Scheme tests on the VM. Usage of this script is described in appendix A.3.

dsvmopt/benchmarks contains all the benchmarks we have used during development and the `runbenchmarks` script to run all the benchmarks on our VM. Usage of this script is described in appendix A.4.

dsvmopt/marks contains the output of several benchmarks taken during development.

ocaml contains some initial exploration of the OCaml language for writing a (efficient) interpreters.

A.1 Building

To build the VM the following software components are required:

- OCaml = 3.10.2
- findlib (ocamlfind) >= 1.2.1
- bitstring >= 1.9.7
- GNU Toolchain (make, gcc and so on)

These tools are all installed on the DAIMI system. To make the OCaml related tools available you must modify a few environment variables. Make sure to add the new paths at the start of the environment variables as they must shadow over already installed OCaml components.

- add `/users/contrib/ocaml/bin` to `PATH`
- add `/users/contrib/ocaml/lib` to `LD_LIBRARY_PATH`

Now, from within the `dsvmopt` directory, type

```
make
```

After that the binary `dsvmopt` should be available in the directory of the same name.

A.2 Running

To run the VM you must use a machine with the IA32 architecture. The following software is required:

- Petite Chez Scheme >= 7.0
- DAIMI-Scheme compiler and assembler

From within the `dsvmopt` directory you may compile a DAIMI-Scheme script to the binary VML format with the following commands:

```
chmod +x compile.sh
./compile.sh tests/fib.scm
```

Or alternatively:

```
chmod +x ../daimi-scheme/bin/*
../daimi-scheme/bin/dscomp tests/fib.scm
../daimi-scheme/bin/dsasm  tests/fib.dsa
```

After compiling the program, it can be run with

```
./dsvmopt -jit tests/fib.dsb
```

Notice the file ending `.dsb`. The above requires that you have successfully compiled our VM as described above in A.1. If not, you may use the precompiled binary that is in our source distribution as `dsvmopt/dsvmopt-pre`. Depending on the method used to build the VM, several flags can be accepted by the VM. For a complete list run

```
./dsvmopt -help
```

A.3 Testing

During the development process we have relied heavily on a test suite consisting of 313 DAIMI-Scheme test programs in order to test various corners of the language semantics. This has been a valuable tool when doing incremental changes to the running VM. The tests are located in:

- `dsvmopt/tests`,
- `daimi-scheme/Tests`, and
- `daimi-scheme/DAIMI-Scheme/tests`.

To run the test suite use the `runtests` script. First the script must be made executable with the following command:

```
chmod +x tests/runtests
```

The tests can be run in two modes, with or without dynamic compilation enabled. To run the tests without dynamic compilation type:

```
./tests/runtests
```

To run the test suite with dynamic compilation enabled type:

```
./tests/runtests -jit
```

The test script will run each DAIMI-Scheme program both on a reference implementation and on our own VM and compare outputs in order to determine whether the test run was successful. For each completed test a dot will be printed to stdout. If a test fails a more detailed description of the two test runs will be printed.

A.4 Benchmarking

The benchmark suite consist of Scheme programs from the Chicken Scheme distribution. We have ported 21 of then to DAIMI-Scheme. The programs range in complexity from a simple recursive tower of hanoi algorithm to a full implementation of the DAIMI-Scheme compiler which compiles itself. All benchmarks are placed in `dsvmopt/benchmarks` along with the unsupported Chicken Scheme programs.

The entire benchmark suite may be run using the `runbenchmarks` script. To do this, first make the script executable with the following command:

```
chmod +x benchmarks/runbenchmarks
```

The benchmarks can be run in two modes with or without dynamic compilation enabled. To run the naive interpreter without dynamic compilation use the following command:

```
./benchmarks/runbenchmarks
```

To run the benchmarks with dynamic compilation enabled type:

```
./benchmarks/runbenchmarks -jit
```

For each completed benchmark program a dot will be printed to stdout. By default benchmark results will be written to `benchmarks/benchmark.txt` but it is possible to supply an alternative output file with the following command:

```
./benchmarks/runbenchmarks -jit benchmarks/some-file.txt "Short Name"
```

The "Short Name" string will supply a name to the benchmark output which is useful when comparing different runs later on. Multiple benchmark outputs can be compared using the following command:

```
make parsebenchmarks
./parsebenchmarks file1.txt file2.txt ...
```

This will print a nice listing of the input benchmark files to stdout, comparing the running times of each individual benchmark program.

B DAIMI Scheme

The DAIMI-Scheme language is a proper subset of the Scheme language (R. Kelsey, 1998) as defined in the R5Rs Scheme language specification. The language was designed for educational purposes and was used in the compiler course at DAIMI at the University of Aarhus in the period 19___-2003 Further documentation can be found along with all the course material at the following location on the daimi NFS:

```
/users/courses/dOvs/Project03/
```

or as part of our project tarball in `daimi-scheme`. It is possible to find the source code for a DAIMI-Scheme compiler and an interpreter for the VML-language, both implemented in DAIMI-Scheme. Furthermore an optimised interpreter written in C exists in binary format. A comprehensive collection of test libraries can also be found.

B.1 VML Assembly Language

The VML assembly language is the intermediate representation of a DAIMI-Scheme program. It comes in two versions: A textual representation with an s-expression syntax and a more compact binary representation. A full description of VML can be found at

B.2 Virtual Machine Registers

Section on VM registers as defined by the DAIMI-Scheme specification. Original text is located in `daimi-scheme/04DAIMI-Scheme-virtual-machine.txt`.

Control registers:

- ip: instruction pointer
- cont: continuation

Environment registers:

- env-lib: vector holding the values of the predefined variables
- env-glo: vector holding the values of the global variables

- env-lex: list of vectors holding the values of the lexical variables
- env-tmp: vector holding the values of the temporary variables

Auxiliary registers:

- aux-res: vector holding the value(s) last returned from a call
- aux-vec: vector for an extension of the lexical environment

Registers are implemented as pointers to vectors:

- ip is assigned during the fetch-decode-execute loop of the byte-code interpreter;
- cont is assigned during the execution of a call instruction (but not during the execution of tail-call instruction) and read & re-assigned during the execution of a return instruction; in addition, it is read when call/cc is called and re-assigned when continuation closures are applied;
- env-lib is initialized to a vector of values, and then it never changes (but the vector entries may change);
- env-glo is initialized to a vector of values, and then it never changes (but the vector entries may change);
- env-lex is frequently assigned: it is saved and initialized when a closure is activated (in call and tail-call), it is modified during the execution of an extend instruction, and it is restored during the execution of a return instruction;
- env-tmp is assigned during the initialization of the VM; it denotes a vector whose entries are frequently assigned;
- aux-res is assigned during the initialization of the VM; it denotes a vector that, in the current specification of DAIMI-Scheme, is always of length 1; this entry is always assigned before executing a return instruction;
- aux-vec is assigned with fresh vectors from the heap.

The vectors pointed to by the registers are:

- a vector holding the values of the predefined variables (held in env-lib);
- a vector holding the values of the global variables of the program (held in env-glo);

In addition, there is also:

- a table of lambdas containing their arity and a pointer to their entry point, initialized with those declared in the program;
- a table of symbols initialized with those declared in the program;
- a table of strings initialized with those declared in the program.

NB. Symbols and strings are specified as part of the load instruction in the assembly format. The binary format, however, has them factored in tables (see the specification of the load instruction below).
The lists pointed to by the registers are:

- a list of vectors of values, one for every lexical-environment extension (held in env-lex);
- a list of activation records, one for every call (held in cont).

An activation record consists of

- a list of activation records (a previous value of cont);
- a list of environment extensions (a previous value of env-lex);
- a pointer to an instruction (a future value of ip); and
- env-tmp[0], ..., env-tmp[n], for some n.