# A Tractable Native-Code Scheme System

Martin Gasbichler
Universität Tübingen
gasbichl@informatik.uni-tuebingen.de

Richard Kelsey
Ember Corporation
kelsey@s48.org

Michael Sperber
sperber@deinprogramm.de

## Abstract

This paper describes a simple and modular approach for adding optimizing native-code compilation to Scheme 48, a byte-code implementation of Scheme. The novelty of the approach is its reuse of existing infrastructure, which enabled incremental and modular development. Each part of the compiler is independently useful. We developed the translator from byte code to native code incrementally, throwing back to the virtual machine for any aspect we had not yet implemented. Moreover, the translator is extremely simple, relegating all serious optimization to a separate, optional pass optimizing the byte-code and producing byte code as a form of portable assembly code. The optimization pass, in turn, was built using a preexisting general-purpose transformational compiler, which also serves to compile the virtual machine source code (written in a subset of Scheme) to highly efficient C. Our approach—using existing, general infrastructure, modular structure, and incremental development of the translation to native code—allowed us to produce a working implementation with comparatively little effort. The implementation achieves significant performance gains over both byte code and unoptimized native code.

## 1. Introduction

Scheme 48 [18] is a byte-code implementation of Scheme [16] constructed with tractability and reliability as its primary design goal. Its clear, modular structure, limited complexity and readable source code make it an ideal platforms for experiments in programming-language implementation. The work described in this paper extends Scheme 48 by a simple native-code compiler.

Of course, the primary goal in implementing native code in a byte-code system is improved performance. However, it was important to preserve the tenets of the existing code base—its simplicity and modular structure—at the same time. This requirement was not for purely esthetic or academic reasons: Our time for implementing the system was limited and non-contiguous, as were the people working on the code. Previous attempts to add native code to the system had failed primarily because of excessive complexity, which prevented that somebody else could take over maintenance from the original developers.

The keys to the ultimately successful effort described here were the following:

- The compiler from byte code to native code is a mere translator from the byte-code instructions to corresponding sequences of native code, and thus extremely simple.
- Optimization is a separate pass that operates on the byte code.
- Existing components of the system were reused.

The last bullet deserves special attention:

The optimizer reuses the transformational compiler that normally compiles the virtual machine (VM). As the optimizer operates on the byte code, optimizations can be tested without involving the native code. No new intermediate representations needed to be developed, as is often the case in byte-code-to-native-code compilers. Instead, the optimizer translates the byte code to the CPS representation of the transformational compiler, optimizes on that representation, and then translates back to byte code. For this to work, it is crucial that the existing byte-code compiler of Scheme 48 is extremely simple and generates very regular code (another consequence of the overall design principle of simplicity), that lends itself to reconstituting the high-level structure of the original Scheme code straightforwardly.

The simplicity of the native-code translator enables tight integration between the byte and the native code. In particular, the native code can always throw back to the VM for any parts not yet implemented, or for run-time aspects of the VM's operation: The native code compiler simply skips code with instructions it cannot translate yet. Changes to a subset of the instructions can be tested without specifying native-code compilators and even without running the optimizer.

The simplicity of Scheme 48's byte-code compiler has already been exploited to obtain a run-time code-generation facility for byte code. The same is possible with the native-code compiler, and composing the byte-code combinators with native-code generation yields native-code run-time code generation. Consequently, these principles lead to a system architecture that keeps Scheme 48's traditional tractability, while making it easy to test and extend the system.

Summarizing, the contributions of our paper are the following:

- We show that a simple system architecture enables adding a native-code compiler to a byte-code system with comparatively little effort.
- A general, tractable infrastructure for compiling and running code enables significant code reuse.
- The resulting system achieves considerable performance improvements over the pure byte-code system, or pure non-optimizing native-code compilation.

***Overview*** Section 2 reviews the overall structure of the Scheme 48 system, including the components involved native code compilation. It highlights the opportunities for reuse afforded by the architecture. Section 3 gives a very brief overview of the architecture of the Scheme 48 virtual machine. Section 4 describes changes made to the original VM to make native-code generation easier and enable faster code. The byte-code optimizer is described in Section 5. Section 6 describes the native-code compiler. Benchmarks are in Section 7. Section 8 briefly describes how we have obtained native-code run-time code generation from the components previously developed. Section 9 reviews some related work, Section 10 discusses directions for future work, and Section 11 concludes.

## 2. Component reuse in Scheme 48

This section gives a rough overview of the Scheme 48 component architecture, emphasizing the places where code reuse happens. Details on the operation of the new components—the byte-code optimizer and the native-code compiler—follow in later sections.

Figure 1 gives an overview over the system architecture of Scheme 48 [18]. Dotted arrows indicate code reuse. The system began as a classic byte-code-based Scheme implementation with a compiler from Scheme to byte code and a virtual machine (VM) that interprets the byte code. However, the VM implementation differs from many other VMs in that it itself is written in Scheme, specifically in a subset called Pre-Scheme.

The VM can run as a normal Scheme program, and VM developers debug the system in that mode of operation. However, the VM can also be compiled to a highly efficient C program by a special compiler that ships the system—the Pre-Scheme compiler, shown on the left. The Pre-Scheme compiler is an excellent illustration for the theme of code reuse that runs through the system and this paper: Its front end simply invokes the regular Scheme front end of the byte-code compiler, which includes the Scheme 48 module system and the macro expander. The central part of the Pre-Scheme compiler is essentially the Transformational Compiler [17] of the T project [19]. The VM, the Pre-Scheme compiler, and the byte-code compiler have been part of the system for a long time. The byte-code optimizer and the native-code compiler are new.

Note that the Scheme front end used in the Pre-Scheme compiler as well as the Transformational Compiler itself are not specific to Pre-Scheme, but are part of a general compiler infrastructure for Scheme. Its distinguishing feature is the use of a single CPS-based intermediate representation for almost all stages of compilation. The Transformational Compiler performs significant optimizations—among them significant partial evaluation, matching up procedures and calling points, and pattern-directed code simplification.

Another component that is reused several times is the byte-code parser. It receives a byte-code code vector and a set of attribution functions, essentially one for each opcode, and calls these attribution functions with the arguments of the opcodes, passing along a state containing an arbitrary value. The byte-code parser learns about the instruction format from a description that also guides the instruction decoding in the VM. In the standard system, the disassembler uses the parser to print a human-readable representation of the byte-code. The new optimizer uses the parser in its front end. The native-code compiler is actually nothing more than a set of attribution functions for the parser.

The byte-code optimizer works by translating the byte code into the intermediate representation of the Transformational Compiler, using the Transformational Compiler to perform the actual optimization, and then turning the CPS code back to byte code. Again, a significant part of the system is reused in a different context.

The native-code compiler is new code, but is extremely simple. It compiles the byte-instructions directly to native code, and performs no analysis whatsoever. It reuses significant code from the VM, in particular the protocol converter for procedure calls and returns, and various other primitive operations. More importantly, it can throw back to the VM for the handling of exceptions and instructions not (yet) handled by the native-code compiler. The latter aspect was especially useful during development and allowed us to run substantial programs in native code before the native-code compiler even handled the full byte-code instruction set.

Figure 1 does not show how the source code of the byte-code compiler was reused to generate combinators for run-time generation, and how the source code of the native-code compiler can similarly be reused to obtain run-time code generation for native code. This is not the main thrust of this paper, however. A brief overview is in Section 8.

## 3. The virtual machine

The Scheme 48 virtual machine interprets a simple stack-based byte-code instruction set.

### 3.1 Data representation and storage management

The tagged data representations are fairly standard for a Scheme system: The system uses word-size descriptors to represent objects. A descriptor can be a fixnum (representing a 30-bit integer), an immediate object (a boolean or character value or the empty list), or a stob descriptor, representing the address of a heap object. Heap objects can be either all-binary or consist entirely of descriptors; their header words contain type and size information.

Scheme 48 offers a choice of two garbage collectors, both of which are precise and moving. The default collector is simple two-space copier; the alternative is a more sophisticated and faster generational BIBOP-based collector whose design is based on Chez Scheme's GC [12]. Details of the VM data representation are described elsewhere [18].

### 3.2 Continuation frames

For managing the continuation frames used in implementing procedure calls, Scheme 48 uses the incremental stack/heap strategy [9]: Continuations are initially created on a stack cache (we will use "stack" for short for the remainder of this paper); they are copied to the heap on overflow or on capture via `call-with-current-continuation`, and copied back on demand by invocation.

The local variables of a procedure are kept on the stack. (The byte-code compiler performs a standard assignment elimination to replace assignment to local variables by operations on heap-allocated cells.) Also, operands and intermediate results from simple primitive applications are kept on the stack.

### 3.3 Byte-code representation

The relevant data structures involved in byte code are closures, templates and environments. All are heap-allocated. A closure is a standard run-time representation of a procedure, consisting of pointers to a template and an environment. The *template* is an artificial object containing a pointer to the actual byte-code object as well as literal values, debug data, and references to global variables used by the procedure. The environment contains all non-local variables.

### 3.4 Instruction set

The VM instruction set mainly operates on the stack. It has a single "accumulator" register `*val*` that typically contains an argument too, and receives the result of a byte-code instruction. For example, the `+` byte-code instruction accepts one argument on the stack, another argument in `*val*`, and also leaves the result in `*val*`.

The VM also keeps a `*code-pointer*` register for the program counter, a `*stack*` register that points to the top of the stack, and a `*cont*` register to remember where the current continuation frame starts on the stack. If the code associated with a closure needs access to the environment and/or the template, these are pushed

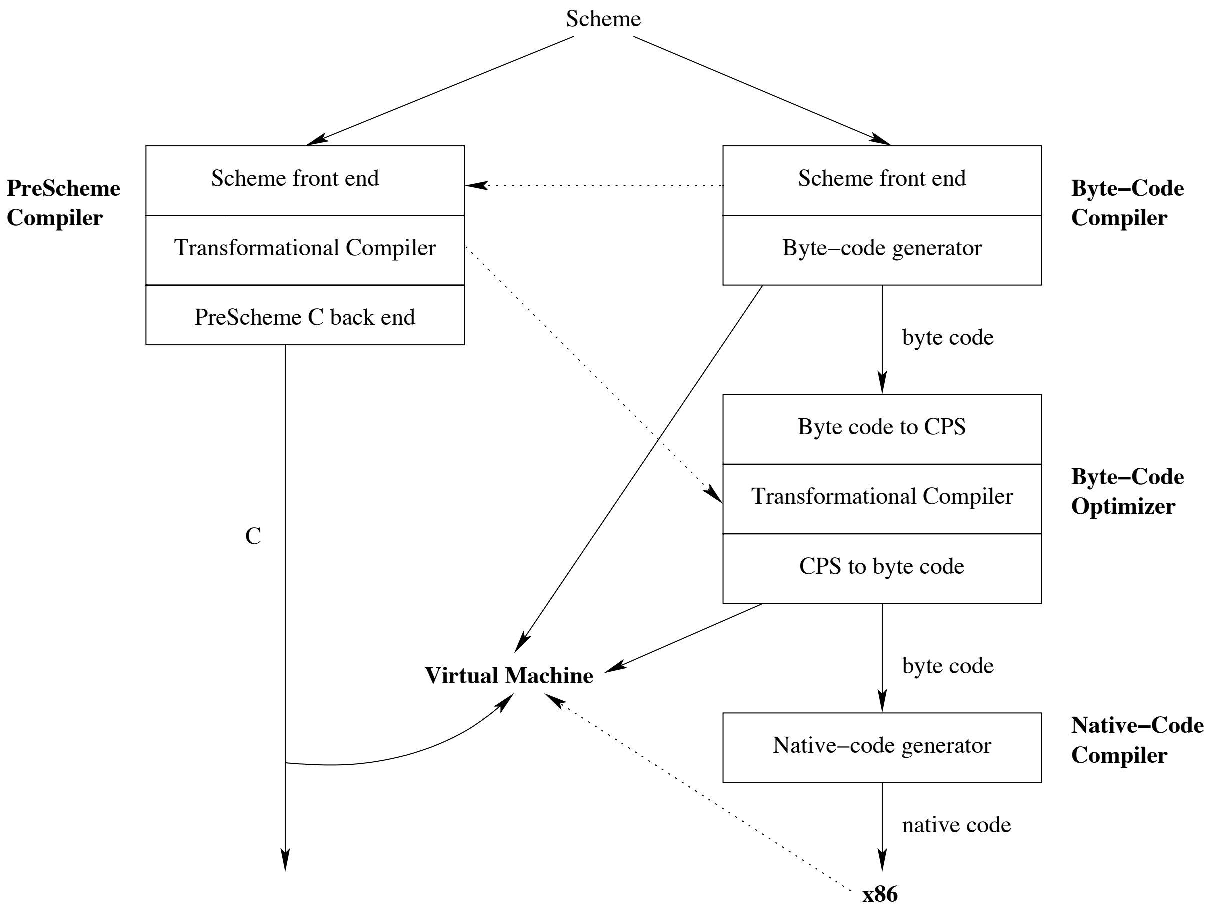


**Figure 1.** Architecture of the Scheme 48 system

on the stack on entry to the procedure. The byte code then uses a `stack-indirect` instruction to refer to individual values within the template or environment.[1]

## 3.5 Procedure calls

Every procedure and continuation includes a protocol marker indicating how it is to be invoked. Mostly this has to do with the number of arguments expected and were they are stored. When invoking a procedure or continuation the invoking code checks that the procedure or continuation has the expected protocol. If the protocols match, the procedure or continuation is invoked directly. If they do not match, the invoking code jumps to a protocol conversion routine that makes any adjustments needed or, if the protocols are inherently incompatible, signals an error. The goal is to make the common case, where the protocols match, fast, while still accommodating the uncommon cases. Most of the complexity in the protocol converter has to do with managing lists of arguments and calling or returning between byte code and native code.

## 3.6 Interrupts

Interrupts are events that the VM asynchronously signals to the run-time system by invoking an interrupt handler. Interrupts correspond to signals sent by the operating system, completion of I/O transactions, or completed garbage collections. A VM register contains the set of pending interrupts as a bit mask. To guarantee that an interrupt is handled within finite time, the VM checks on every call if there is a pending interrupt. To speed up this test, it is combined with the check for sufficient stack space also required upon each procedure call: code that sets a bit in the interrupt mask also sets the stack limit to `-1`. Then the next stack space test will fail and the handler can recognize that actually an interrupt is pending. Checking on every procedure call is sufficient in the case of standard byte code as this code contains no backwards jumps. However, the optimizer must insert `poll` instructions as it may produce loops. A `poll` can also simply check if the stack limit is `-1` to learn that there are pending interrupts.

[1] Earlier versions of the virtual machine [18] also had registers for the current template and the current environment.

## 3.7 Exceptions

The VM maintains a vector of exception handlers, one for each opcode. The RTS may set this vector using a special opcode. When the VM detects a failure executing an opcode, it sets up an exception continuation and continues the interpretation with the code of the corresponding exception handler. The exception continuation continues the execution with the instruction following the failure.

## 4. Streamlining the VM

The original VM architecture used linked environments allocated on the stack and then copied to the heap when a closure was created. The linked environments followed the lexical structure of the program. This avoided the need for doing any live variable analysis in the compiler.

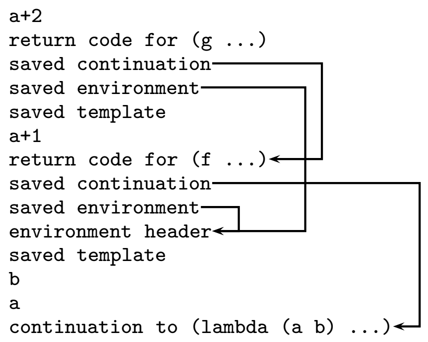
The use of the stack followed directly from the source code. The top of the stack was used for ordinary stack execution of expressions, with call arguments being pushed from right to left. Once the arguments were complete, the procedure would be invoked and would push its own lexical environment and a header onto the arguments, forming them into an environment. For a non-tail call, the values of the registers and a header were pushed on top of the current value stack, forming a continuation. The stack thus contained a set of continuations and environments, with links that always pointed down the stack. The stack would grow until a continuation was invoked. A series of tail-recursive calls would eventually fill the stack with unreachable environments. Stack space was reclaimed by copying the live environments and continuations to the heap using the same mechanism as for `call-with-current-continuation`. Consider the following example:

```
(lambda (a b)
    (+ (f (+ a 1) (g (+ a 2)) (+ b 2))
       1))
```

With the arguments pushed from right to left and no reordering of calls, the stack contains the following values before the call to g:

```
a+2
continuation for (g ...)
a+1
continuation for (f ...)
b
a
continuation to (lambda (a b) ...)
```

The following intra-stack pointers appear:



```
a+2
return code for (g ...)
saved continuation
saved environment
saved template
a+1
return code for (f ...)
saved continuation
saved environment
environment header
saved template
b
a
continuation to (lambda (a b) ...)
```

Later, live variable analysis was added for determining the precise environments needed for closures, along with the introduction of explicit cells for `set!` variables. Closure environments were then flat vectors, with no indirection. Once explicit cells were introduced, it was no longer necessary for closures and continuations to share the same environment structure. This avoided the need for intra-stack pointers for environments. The architecture was also changed to delay the creation of continuations until the time of the call. Space was still reserved for the return code pointer, but the pointer itself was not added until immediately before the call. This maintained the frame as a single contiguous object and avoided the need for intra-stack pointers to continuations. Continuations were implicitly linked by being adjacent in the stack. Some other minor changes were also required, including pushing the called procedure's environment and template on the stack instead of in VM registers (which are not shown in the stack diagrams).

At the time of the call to g, the stack would now appear as:

```
a+2
return code for (g ...)
a+1
nil (saved space for the return code for (f ...))
template
b
a
continuation to (lambda (a b) ...)
```

To maintain the adjacency of continuations, arguments to tail-calls were copied down over the current stack frame before invoking the tail-called procedure. This is less efficient than the previous stack GC, but more than outweighed by the gains from reducing the number of intra-stack links that need to be created and dereferenced.

These changes allowed the unoptimized and optimized byte codes to use the same stack organization, although not necessarily identical frames. The byte-code optimizer makes more efficient use of the stack by not intermixing the code for nested calls. For the optimized code, the stack at the time of the call to g would look like this:

```
a+2
return code for (g ...)
template
b
a
continuation to (lambda (a b) ...)
```

Using the same stack organization for unoptimized and optimized byte code, and thus for native code as well, avoided the need to modify the runtime code, such as the debugger, when adding the native code compiler.

## 5. The byte-code optimizer

The byte-code optimizer works basically in three stages:

1. a front end translates the byte code by to the CPS intermediate representation of the Transformational Compiler,
2. a series of optimizations, all operating on the intermediate representation,
3. a back end that turns the intermediate representation back into a byte code.

The following subsections expand, respectively, on the intermediate representation, and the three stages of the optimizer itself.

### 5.1 CPS intermediate representation

The CPS intermediate language used in the compiler consists of the $\lambda$-calculus with the addition of constants and annotations on $\lambda$ forms. The constants include data constants, such as integers and strings, and primitive operators (*primops*). Note that even regular procedure calls and returns are explicit primops; the explanation below has details. The annotations on the $\lambda$ forms divide the forms into three classes: continuations, jump targets, and procedures, depending on how the value of the form is used. Continuations are $\lambda$ forms passed as continuation arguments to primops. Jump targets are $\lambda$ forms whose calling points have been identified and which are all within the same procedure $\lambda$. All other $\lambda$ forms are procedures.

Within the compiler CPS programs are represented as a tree containing four types of nodes: lambdas, calls, literals, and variable references. As described above, lambda nodes are further divided into $\lambda_{\mathrm{cont}}$, $\lambda_{\mathrm{jump}}$, and $\lambda_{\mathrm{proc}}$ nodes according to their usage. All calls are to primitive operators (primops), and are either trivial, if the primop simply computes a value, or nontrivial, if the primop has side effects or directly affects the control flow. Calls to nontrivial primops have continuation arguments, calls to trivial primops do not. Calls to procedures are represented as calls to primops that invoke one of their arguments.

The node tree has a very regular structure:

- The body of every lambda node is a non-trivial call.
- The parent of every non-trivial call is a lambda node.
- Every cont lambda is a continuation of a non-trivial call.
- Every jump lambda is an argument to either the `let` or `letrec` primops (described below).

- The lambda node that binds a variable is an ancestor of every reference to that variable.

A basic block appears as a sequence of non-trivial calls each with a single continuation that links the sequence. The block begins with a $\lambda_{\text{proc}}$ or $\lambda_{\text{jump}}$, or with a $\lambda_{\text{cont}}$ that is an argument to a call with two or more continuations, and ends with a call with no continuations, such as a return or tail-call, or a call to a conditional primop, which get two or more continuations.

Basic blocks are grouped into trees. The root of every tree is either a proc or jump lambda, the branch points are calls with two or more continuations, and the leaves are jumps or returns. Within a tree the control flow follows the lexical structure of the program from parent to child (if we ignore calls to other $\lambda_{\text{proc}}$s).

Every $\lambda_{\text{jump}}$ is called from within only one $\lambda_{\text{proc}}$, so for control flow a $\lambda_{\text{proc}}$ can be considered to consist of a set of trees, the leaves of which either return from that $\lambda_{\text{proc}}$ or jump to the top of another tree in the set.

For the following five primops the $\lambda$ node being called, jumped to, or whatever has been identified by the compiler, and the number of variables that the lambda node has matches the number of arguments.

- `(call` *cont proc* `.` *args*`)`
- `(tail-call` *cont-var proc* `.` *args*`)`
- `(return` *cont-var* `.` *args*`)`
- `(jump` *jump-var* `.` *args*`)`

The next three primops are the same as the above except that the being called procedure has not been identified by the compiler. There is no `unknown-jump` primop because all calls to $\lambda_{\text{jump}}$s must be known.

- `(unknown-call` *cont proc* `.` *args*`)`
- `(unknown-tail-call` *cont-var proc* `.` *args*`)`
- `(unknown-return` *cont-var* `.` *args*`)`

$\lambda_{\text{proc}}$ nodes are called with either `call` or `tail-call` if all of their call sites have been identified, or with `unknown-call` or `unknown-tail-call` if not. $\lambda_{\text{jump}}$ nodes are called using `jump`.

The primop `let` binds values such as lambda nodes or the results of trivial calls to variables. This primop is needed because of the requirement that every call have a primop; all it does is apply *cont* to *args* (it is called `let` instead of `apply` because `let` forms in the source code become calls to this primop).

- `(let` *cont* `.` *args*`)`

Recursive binding is implemented using a pair of primops.

- `(letrec1` *cont*`)`
- `(letrec2` *cont id-var lambda1 lambda2* `...)`

These are always used together, with the body of the continuation to `letrec1` being a call to `letrec2`. The two calls together look like:

```
(letrec1 (lambda/cont (id-var var_0 ... var_n)
           (letrec2 cont id-var lambda_0 ... lambda_n)))
```

This binds $var_i$ to $lambda_i$, with the $lambda_i$ in the lexical scope of the $var_i$. The *id-var* is used to verify that matching `letrec1` and `letrec2` calls have not gotten separated by a misbehaving compiler transformation.

- `(test` $cont_t$ $cont_f$ *arg*`)`

This primop is for conditionals; it accepts two continuations $cont_t$ and $cont_f$, and the value of *arg* determines which one gets invoked: $cont_t$ for a true value, $cont_f$ for false.

## 5.2 Byte code to CPS

The converter from byte code to the CPS intermediate representation accepts a closure and turns it into a CPS $\lambda_{\text{proc}}$ node. The conversion depends on the regular way in which the code generated by the byte-code compiler uses the stack, and uses a fairly standard abstract-stack analysis to keep track of what the individual slots on the stack contain: As the converter passes through a basic block, it maintains an abstract representation of the contents of the `*val*` register and the stack. The abstract representation distinguishes the following cases:

- a literal directly pushed by a byte code,
- a variable in the CPS representation, denoting an intermediate result of the computation previously examined,
- the environment containing abstract value representations, which was pushed as part of the procedure-call protocol,
- the template, which was pushed as part of the procedure-call protocol,
- a reference to the environment of the closure, which needs to be preserved.

All intermediate results on the stack become assigned variables in the CPS node as it is being constructed. Moreover, the converter needs to track targets of jumps, which it turns into $\lambda_{\text{jump}}$ nodes. It also decodes the constructions of recursive environments, which are the compilation product of `letrec` expressions in the original Scheme program.

As an example, consider the following Scheme procedure:

```
(define (tally p l)
  (let loop ((l l) (a 0))
    (if (null? l)
        a
        (loop (cdr l) (if (p (car l)) (+ a 1) a)))))
```

`Tally` takes a predicate and a list and returns the number of elements in the list for which the predicate returns true. The CPS code produced by the conversion is reasonably close to the original:

```
(P tally_4 (c_3 p_2 l_1)
  (LET* (((x_29 loop_5)
                  (letrec1))
         (()      (letrec2 x_29 ^loop_9)))
    (unknown-tail-call c_3 loop_5 l_1 '0)))

 (P loop_9 (c_8 l_7 a_6)
   (LET* (((v_10)*    (eq? l_7 '())))
     (test 2 ^c_12 ^c_13 v_10)))
  (C c_12 ()
    (unknown-return c_8 a_6))
  (C c_13 ()
    (LET* (((v_14)*    (cdr l_7))
           ((v_16)*    (car l_7))
           ((v_19)     (unknown-call p_2 v_16))
           ((j_26)*    ^j_25))
      (test ^c_20 ^c_21 v_19)))
   (C c_20 ()
     (LET* (((v_22)*    (+ a_6 '1)))
       (jump j_26 v_22)))
   (C c_21 ()
     (jump j_26 a_6))
   (J j_25 (v_24)
     (unknown-tail-call c_8 loop_5 v_14 v_24))
```

The `P` forms stand for $\lambda_{\text{proc}}$ nodes, the `C` nodes stand for $\lambda_{\text{cont}}$ nodes, and the `J` nodes stand for $\lambda_{\text{jump}}$ nodes. (The indentation indicates lexical depth.) The `LET*` expressions describe both calls to `let` primop (marked with a `*` after the bound variables), as well as calls to primops with a continuation. Note that the latter may bind several variables. Thus, the first two bindings in `c_13` are really calls to `let`, and the third is a call to the `unknown-call` primop, with a continuation that binds `v_19`. The `letrec1` and `letrec2` calls at the top are really nodes of the following form:

```
(letrec1
  (lambda/cont (x_29 loop_5)
    (letrec2
      (lambda/cont ()
        (unknown-tail-call c_3 loop_5 l_1 '0)))
    x_29 ^loop_9))
```

References like `^loop_9` indicate that really the node of that name, which occurs later in the printout, appears here.

## 5.3 Optimization

The transformational compiler applies a variety of aggressive optimizations to the CPS nodes that the converter produces, among them constant folding, beta reduction, matching up procedures and calling points, turning recursive loops into iterative ones, and boolean short circuiting.

For the `tally` example, the optimizer does not have much to do due to its small size; as the continuation argument of the `loop_9` $\lambda_{\mathrm{proc}}$ is `c_3` for both calls, the optimizer specializes `loop_9` for `c_3`. The $\lambda_{\mathrm{proc}}$ then becomes a $\lambda_{\mathrm{jump}}$ because its continuation is gone and the two corresponding applications of `unknown-tail-call` into `jump`s:

```
(P tally_4 (c_3 p_2 l_1)
  (LET* (((x_34 loop_5)
                    (letrec1))
         (()        (letrec2 x_34 ^loop_9)))
    (jump loop_5 l_1 '0)))

 (J loop_9 (l_7 a_6)
   (test ^c_12 ^c_13 (eq? l_7 '())))
  (C c_12 ()
    (unknown-return c_3 a_6))
  (C c_13 ()
    (LET* (((v_14)*    (cdr l_7))
           ((v_19)     (unknown-call p_2 (car l_7))))
      (test ^c_20 ^c_21 v_19)))
   (C c_20 ()
     (jump loop_5 v_14 (+ '1 a_6)))
   (C c_21 ()
     (jump loop_5 v_14 a_6))
```

## 5.4 Byte-code annotation

The rest of the byte-code optimizer is concerned with turning the CPS code back into code. The first task at hand is the re-introduction of template references (for literals and subtemplates representing internal procedures) and explicit closures and environment references as required by the byte code (see Section 3.3). Both are straightforward. The example involves only a template reference:

```
(P tally_4 (c_3 e_36 t_37 p_2 l_1)
  (LET* (((x_34 loop_5)
                    (letrec1))
         (()        (letrec2 x_34 ^loop_9)))
    (jump loop_5 l_1 '0)))

 (J loop_9 (l_7 a_6)
   (test ^c_12 ^c_13 (eq? l_7 (template-ref t_37 '4))))
  (C c_12 ()
    (unknown-return 0 c_3 a_6))
  (C c_13 ()
    (LET* (((v_14)*    (cdr l_7))
           ((v_19)     (unknown-call p_2 (car l_7))))
      (test ^c_20 ^c_21 v_19)))
   (C c_20 ()
     (jump loop_5 v_14 (+ '1 a_6)))
   (C c_21 ()
     (jump loop_5 v_14 a_6))
```

The parameter `e_36` is newly introduced for the implicit environment of the closure, and `t_37` is the template. The one template access is for the literal access to the empty list. In the example, `e_36` is dead and will actually be eliminated in the final output code.

The re-introduction of environments takes the opportunity of merging the closure and environment objects for all non-recursive objects—the values of the free variables occur in the closure object itself, rather than an environment pointed to by the closure.

Just before the actual code generation, the byte-code optimizer allocates stack offsets to variables. Every live variable is assigned a location in a stack frame. This is done by a simple graph coloring algorithm, which reuses stack frames as the variables they represent become dead.

Furthermore, for tail calls the optimizer has turned into loops, it must also insert explicit polling calls before back edges in the control-flow graph to check for interrupts (see Section 3.6). The example requires no polling instructions, as every iteration of the loop performs a procedure call.

## 5.5 Byte-code generation

With the annotation done by the previous phases, the generation of byte code is again a straightforward process. Whereas the original byte code managed intermediate values by pushing and popping on the stack, the byte code emitted by the optimizer operates within a fixed stack frame allocated at the very beginning.

The main complication arises through back edges in the control-flow graph. These are tail calls in the original byte code, which uses the simple but slow protocol of creating the new stack frame on top of the current one and then moving it downwards the stack. The byte-code optimizer uses a `stack-shuffle` instruction to directly turn the current stack frame into the stack frame needed at the destination of the back edge.

The byte code emitted by the example is as follows (with explanatory comments):

```
 0 (protocol 2 (push template)) ; environment is dead
 3 (push-n 2) ; fixed stack frame
 6 (integer-literal 0)
 8 (stack-set! 1)
10 (jump (=> 13))
13 (stack-ref 3)
15 (push)
16 (template-ref 3 4) ; template is 3 into the stack
19 (eq?)
20 (jump-if-false (=> 26))
23 (stack-ref 1)
25 (return)
26 (stack-ref 3)
28 (stored-object-ref pair 1) ; cdr
31 (stack-set! 0)
33 (push-false) ; placeholder for return address
34 (stack-ref 4)
36 (stored-object-ref pair 0) ; car
39 (push)
40 (stack-ref 6)
42 (call (=> 57) 1)
46 (cont-data (=> 57))
57 (protocol 1) ; continuation accepts one value
59 (stack-set! 3)
61 (stack-ref 3)
63 (jump-if-false (=> 81))
66 (integer-literal 1)
68 (push)
69 (stack-ref 2)
71 (+)
```

```
 72 (stack-set! 1)
 74 (stack-shuffle! 1 (1 4)) ; shuffle stack 0 -> 3
 78 (jump-back (=> 13))
 81 (stack-shuffle! 1 (1 4)) ; shuffle stack 0 -> 3
 85 (jump-back (=> 13))
```

Note that the indices with the `stack-shuffle!` instruction are incremented by one to allow for an additional slot at 0 used to correctly shuffle in the presence of cycles.

### 5.6 Peephole optimizations

The emitted byte code still offers numerous opportunities for peephole optimization—a `stack-set!` followed by a `stack-ref` to the same location is common, likewise jumps to the next instruction. A pattern-directed peephole optimizer operating on the byte code catches the most common cases. Its implementation reuses the byte-code parser in the front end and the standard system assembler to produce the output. The peephole optimizer also re-introduces the peephole instructions of the VM, and combines predicates (producing a Scheme boolean value) followed by conditional jumps into single instructions combining the test and the jump. This is the result for the `tally` example:

```
  0 (protocol 2 (push template))
  3 (push-n 2)
  6 (integer-literal 0)
  8 (stack-set! 1)
 10 (stack-ref+push 3)
 12 (template-ref 3 4)
 15 (jump-if-not-binary (=> 22) eq?)
 19 (stack-ref 1)
 21 (return)
 22 (stack-ref 3)
 24 (stored-object-ref pair 1)
 27 (stack-set! 0)
 29 (push-false)
 30 (stack-ref 4)
 32 (stored-object-ref pair 0)
 35 (push+stack-ref 6)
 37 (call (=> 52) 1)
 41 (cont-data (=> 52) (depth 5) (template 2) (live 0 1 2 4))
 52 (protocol 1)
 54 (stack-set! 3)
 56 (jump-if-false (=> 73))
 59 (integer-literal+push 1)
 61 (stack-ref 2)
 63 (+)
 64 (stack-set! 1)
 66 (stack-shuffle! 1 (1 4))
 70 (jump-back (=> 10))
 73 (stack-shuffle! 1 (1 4))
 77 (jump-back (=> 10))
```

## 6. The native-code compiler

The native-code compiler converts a byte-code closure (optimized or not-optimized) to native code. A set of *compilators*, one for each byte-code instruction, emits separate pieces of native code for each byte-code instruction. A compilator can generate straight-line code directly equivalent to the corresponding byte-code instruction. However, a compilator may also resort to the following external facilities to implement the instruction:

- call static glue code reachable via global labels
- request operations provided by the VM
- restart the VM
- call C functions exported by the VM

The static glue code implements functionality that would take too many instructions in the generated code. A small set of Scheme procedures emit the glue code, which, as detailed in Section 6.2, mediates between the native code and the VM C code. The other facilities in the list are invoked via the glue code. For requesting a operation or restarting the VM, the glue code pushes a return address or continuation frames to enable returning to native code. Sections 6.2 and 6.3 detail these two mechanisms. For calling a C function exported by the VM, the glue code implements the calling convention and the linker fills in the address of the called function. Sections 6.4 and 6.5 contain corresponding examples.

The compiler supports these facilities by providing the compilators with the addresses of the glue code. The restart facility also requires the byte code to still be available. To that end, templates have an additional slot that always contains the byte code; the native-code compiler only replaces the original slot. The run-time system may also use the slot containing the byte code to associate the original debugging information with continuations on the stack.

### 6.1 The driver of the native-code compiler

The driver of the native-code compiler maps the compilators over a byte-code vector and produces a new closure. It also recursively descends into any templates contained in the closure's template. These sub-templates contain the code of locally defined procedures.

To disassemble the byte code, the native-code compiler uses the byte-code parser. The state passed between the compilators is an instruction-stream object into which the compilators emit their code and which an assembler turns into the final code vector. The compiler itself is only about 100 lines of code.

### 6.2 Interfacing VM and native code

The glue code is responsible for switching from byte code to native code and vice versa. Native code and byte code communicate via the `*val*` register, the stack, and other VM registers. The glue code synchronizes `*val*` with `%eax` and the VM stack pointer `*stack*` with `%esp`, and makes the addresses of the other registers available in a global array.

Switching from byte code to native code potentially happens upon procedure calls and returns. This is merely a matter of extracting the pointer to the first instruction and jumping to it. Whenever the native code needs to call a byte-code procedure or return to a byte-code continuation, it simply returns to the VM with a special *service tag* describing the action requested of the VM. This tag gets passed to the `post-native-dispatch` procedure in the VM:

```
(define (post-native-dispatch tag)
  (let loop ((tag tag))
    (case tag
      ((0)
       (goto return-values s48-*native-protocol* null 0))
      ((1)
       (goto perform-application s48-*native-protocol*))
      ((2)
       (let* ((template (pop))
              (return-address (pop)))
         (cond ((pending-interrupt?)
                (goto handle-native-poll template
                                         return-address))
               (else
                (loop (s48-jump-native return-address
                                       template))))))
      ((4)
       (goto interpret *code-pointer*))
      ((5)
       (goto do-apply-with-native-cont s48-*native-protocol*
                                       (pop)))
      (else
       (error "unexpected native return value" tag)))))
```

If the tag is 0, the native code wants the VM to invoke a continuation. This happens if the continuation contains byte code, if a `values` instruction wants to return to byte code, or if the continuation's protocol is too complex or does not match. In all cases, the native code has set `s48_Snative_protocolS` (available to the VM as `s48-*native-protocol*`) to the number of values to return and the top of the stack points to the continuation to be invoked.

If the tag is `1`, the VM invokes a procedure on behalf of the native code. In this case `*val*` contains the procedure and `s48-*native-protocol*` the number of arguments on the stack. The procedure either contains byte code, a complex protocol, or the number of arguments did not match, or the native code detected an interrupt while calling. Thus the native code only implements the most common case of calling a native-code procedure with a small number of arguments. In order to do so, it ensures that `%eax` contains a closure, checks the protocol and the stack limit and then jumps to the code.

The native code sets the tag to `2`, if its implementation of `poll` has detected a pending interrupt. The VM receives the template and the address after the `poll` on the stack and `handle-native-poll` constructs a continuation for the interrupt handler before calling the handler. If the interrupt must be ignored, `s48-jump-native` directly returns to the native code.

If the tag is `4`, the native code failed to execute an opcode and wants the VM to retry. Section 6.3 details the restarting mechanism.

The glue code jumped to by the compilator of `apply` sets the tag to `5`. The compilator and the glue code only need to put the number of stack arguments into `s48_native_protocol` and set up the continuation. The VM procedure `do-apply-with-native-cont` performs the protocol conversion and invokes the procedure. After the protocol conversion, invoking the procedure is simple even for native-code procedures. Note that the compilator of `apply` always returns to the VM and thus needs to emit only four instructions. The glue code is likewise very short.

### 6.3 The restart service

The restart service allows a native-code compilator to let the VM interpret the byte-code instruction the compilator is supposed to implement. This allows a compilator to implement an instruction only partially—say, for a restricted set of arguments—and resort to the VM for the full semantics. Restarting is possible because of the simplicity of the compiler: as it translates one instruction at a time, the VM can take over at any time. Aside from the value register and the stack, which the glue code can synchronize easily, the only other information required by the VM is the byte-code instruction pointer. The glue code can reconstruct the instruction pointer from the template and the byte-code program counter, which is passed to each compilator. However, the compilator must ensure that the VM continues with the native code after it has completed executing the byte-code instruction.

As an example, in case of a failure, the native code requests a restart from the VM, which redetects the failure and raises the exception. The glue code sets up the code pointer and the registers for the VM. (This is only three instructions, which keeps the pressure on the instruction cache low.) The continuation to which the handler returns must jump back to the original native code: The glue code achieves this by generating a native-code continuation and storing a pointer to it in a special variable.

As the glue code copies `%eax` to `*val*` and the VM assumes the remaining non-instruction-stream arguments of an opcode to be on the stack, the native code must not modify these locations before it requests the restart. This sometimes results in the insertion of additional instructions, either to move the contents of `%eax` to another register before modifying it or using stack reference instructions instead of `pop` to access operands combined with a final instruction to pop all arguments at once. Especially for `%eax`, this overhead is low. In most cases, copying would be required anyway, as the code for the type checks are destructive on the x86.

### 6.4 Arithmetic operations

The Scheme 48 virtual machine provides instructions for generic arithmetic directly. However, it only performs fixnum and bignum arithmetic directly, and raises an exception for all other number types. The run-time system handles these cases.

For inline native code, only fixnum arithmetic is practical. (Flonum arithmetic is future work.) However, using restarting to make use of the VM's implementation for bignums would require the VM to check after each arithmetic instruction whether is must continue to interpret byte code or return to native code (see Section 6.3). Instead, the VM exports the floating-point and bignum operations as C functions along with type predicates for them. The glue code uses these functions if the corresponding compilator has detected that the arguments (or the result) are not fixnums.

### 6.5 Other complex opcodes

There are several opcodes where a translation to native code would be impractical. These are opcodes that really provide library functionality, that either implement an interface to the operating system or, in very few cases, are part of the VM to provide a performance advantage over byte code. (For example, providing `read-char` in the VM provides a 10x–15x speedup in I/O-bound code over the equivalent byte-code implementation.) For these opcodes, the compiler takes the an approach similar to the arithmetic operations: the VM makes the corresponding functionality available to the native code as C functions, and the compilators call these functions. The exceptions raised by these functions push exception continuations that jump to the original native code.

### 6.6 Allocation

The native code currently supports allocation of heap objects using the standard two-space copier of Scheme 48. (Support for the new BIBOP collector is planned.)

Allocation gets the address of the new object from the VM register `*hp*`. After saving this address, the code increments the register and compares the result with the limit of the heap. If the limit is not reached, the allocation was successful. Otherwise code in an accompanying floating block will jump to an external label which will undo the failed allocation and trigger a garbage collection. After the requested number of bytes have been freed, the glue code will return to the allocation, where the register can now be loaded successfully.

### 6.7 Optimizations

The generated code still provides various opportunities for improvement on the machine code level. For example, each compilator currently places the result in `%eax` often at the cost of additional `movl` instructions. This can be easily remedied by alternately using a second register and generating a second version of the static glue code tailored to synchronize `*val*` with it. Other simple optimizations include the proper alignment of jump targets and loop unrolling if arguments to compilators are statically known.

## 7. Benchmarks

Figure 2 contains some preliminary benchmarks comparing standard byte code with itself compiled to native code. Figure 3 compares non-optimized with optimized native code, and the PLT MzScheme 352 JIT [13]. The benchmarks are a subset of the standard Gabriel benchmarks [14]. The timings where obtained on a machine 3.00GHz Pentium 4 system, running FreeBSD 5.4 with a heap size of 160MB. The MzScheme timings do not include garbage collection, because the Scheme 48 heap is so large that almost no GCs happen there.

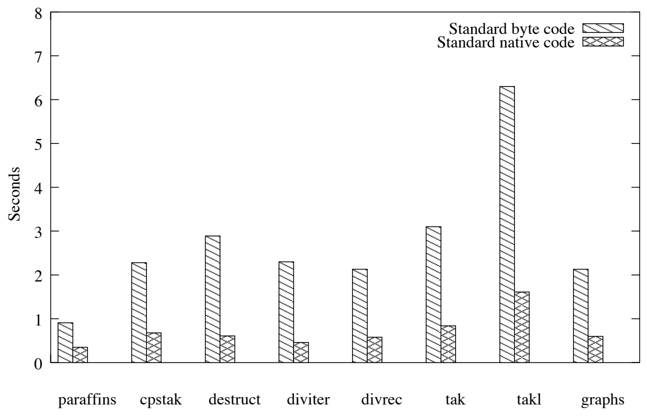


**Figure 2.** Timings comparing byte code and native code

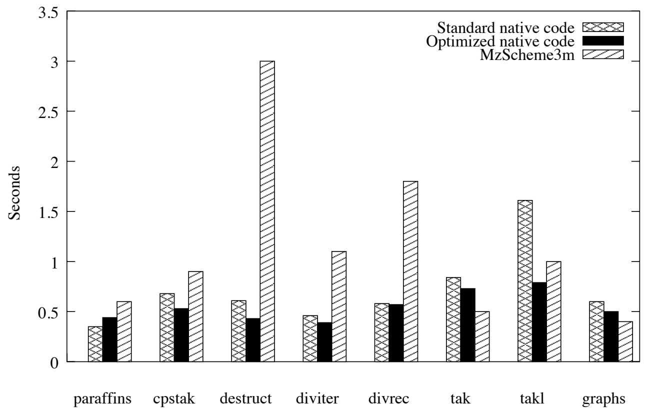


**Figure 3.** Timings comparing non-optimized native code, optimized native-code, and MzScheme

The benchmarks show that the native-code compiler achieves a speedup between 2 and 4 and that the optimized native code can add an additional speedup up to a factor of 2, especially if the optimizer is able to turn tail-calls into jumps. The gains are low if there are lots of recursive calls probably because these are currently not turned into direct jumps but the processor must do an indirect `jmp`. We are currently investigating the case where the optimized code is slower that the original code and we are confident to remedy this soon. The comparison with MzScheme indicate that our approach is already able to compete with other optimizing Scheme implementations, even though there is still much room for improvement (see Section 10).

## 8. Run-time code generation

Sperber and Thiemann have developed a facility for implementing run-time code generation (RTCG) for byte code in Scheme 48[23]. This infrastructure relies on generating combinators for code generation from the source code of the byte code compiler itself. These combinators are then symbolically composed with the back end of a partial evaluator, which yields the RTCG facility. Hence, the generators created by the partial evaluator directly generate byte-code without creating intermediate source-code data structures, which is significantly more efficient than generating and then compiling source code. This is another example of reuse in the Scheme 48 infrastructure. It crucially depends on the simplicity of the byte-code compiler, which acts as a catamorphism on the source code.

With the native-code compiler in place, we have composed Sperber's and Thiemann's byte-code-generation generators from the RTCG facility with the compilers of the native-code compiler, and thus achieved straight-through native-code code generation, with no intermediate source or byte code, at very little additional cost, again almost completely relying on reused code.

## 9. Related work

The advent of Java and the Java Virtual machine led to a large body of research and implementation work on just-in-time compilers. Even though much of that work is relevant to our subject area, there is too much of it for us be exhaustive or fair in our selection. Aycock's survey paper [5] gives an extensive historical overview.

Because of the large demand for fast execution of JVM code, many high-performance JVMs perform different levels of sophisticated optimizations, directed by profile-driven analysis. Most code-generation engines for the JVM either involve one or several intermediate representations between the JVM code and native code, and are thus significantly more complex than our approach. Even those JITs that feature one-pass translation of byte code to native code perform rudimentary analysis, typically involving at least an abstract stack analysis similar to that performed by our byte-code optimizer.

Sun's original JIT described by Cramer et al. [10] compiles straight-line JVM code to native code, but delays generating code that evaluates expressions until the result is needed, reconstituting the original expression along the way and generating special code only then. Adl-Tabatabai et al. [2] describe a JIT developed by Intel that similarly translates directly to native code, calling their low-overhead code-generation *lazy code selection*, but even their technique involves an abstract-stack analysis, and the construction of a control-flow graph for register allocation.

The code generators of the sophisticated JITs that are part of Sun's HotSpot system [21], IBM's Java Development kit [24], and the Jalapeño VM [3] (now Jikes) are all considerably more complicated and involve multiple intermediate representations.

The virtual machine of the Squeak implementation of Smalltalk [15] is constructed analogously to the Scheme 48 VM: It is written in a subset of Smalltalk that is translated to C code.

PLT MzScheme [13] is another originally byte-code-based Scheme implementation that has recently gained a JIT compiler, which achieves significant performance improvements. MzScheme's JIT is part of the core (written in C) implementation of its virtual machine, and uses GNU Lightning for code generation.

Many Scheme, Lisp, and ML systems have traditionally featured dynamic compilation to native code as their normal mode of operation [22, 4, 20, 8, 11]. However, most of these systems skip the generation and interpretation of byte code altogether. If they do supply an „interpreted mode", they usually interpret at a higher representational level than byte codes. The MacScheme system generated native code not directly from byte code, but from a representation similar to it [6]. MIT Scheme [1] also compiles from byte code to native code, but the compilation is offline and file-based.

On of the authors was previously successful implementing a compiler from byte code to PowerPC native code in his Master's thesis. His work showed that a simple translator from byte code to native could be implemented with limited effort, and still significant performance improvements of factors usually between 2 and 4. However, his results also showed that native-compilation could benefit from changes in the VM architecture, and that more optimization would be needed to achieve the performance customarily associated with native-code compilation.

## 10. Future work

There are several areas where the system still has some way to go:

***Optimizations*** A number of relatively simple optimizations should considerably improve the quality of the output code, specifically:

- lambda lifting [8],
- table-driven case dispatch [7],
- optimization of the parallel assignment code to drop procedure arguments into the right stack slots immediately, and thus minimizing the work `stack-shuffle` has to do.

Furthermore, a coworker is working on a flow-analysis framework to enable flow-directed inlining, removal of redundant tag checks, and unboxing optimizations. (The CPS representation of the transformational compiler is eminently suitable for this task.)

***JIT*** The native-code compiler currently runs as regular Scheme code on top of the VM, but should really run transparently as part of the VM's operation. To this end, the compiler needs to be translated to Pre-Scheme.

***Fancier native-code generation*** Native-code generation could presumably benefit from a more detailed analysis of the byte code. In particular, the fixed stack frame produced by the byte-code optimizer could be a good starting point for register allocation. We remain doubtful whether this line of work will produce results that justify the increase in complexity.

***Profile-driven compilation*** As with current byte-code JIT implementations (see the section on related work), the operation of the byte-code optimizer and the native-code compiler should be directed by a profile-driven analysis of the running code.

***Multi-processor support and Kali Scheme*** A coworker has already added multi-processor support to Scheme 48, which works by starting a variant of the VM (called *virtual processor*) in a POSIX thread. Combining the native-code support with virtual processors should be a mere matter of adapting the glue code. Likewise, a coworker has Kali Scheme, the distributed variant of Scheme 48, to the current development version. Adding our native-code compiler to Kali should be as simple as replacing native code by (optimized) byte code within templates before transmitting over the net and invoking the compiler upon receiving a template.

## 11. Conclusion

We have shown how modular design with simple components enable significant code reuse in a programming language implementation. This reuse has transcended several generations of projects and programmers. Therefore, these reuse opportunities are more than a mere accidents. The crucial aspect has been the simplicity in the design of the individual components, which has enabled their composition. Thus, tractability is useful not only as a goal in and of itself, but is rather an indispensable property of the system as it is being extended.

The byte-code optimizer improves on the results of the byte-code compiler, and relies on the existing transformational compiler for the optimization proper. The optimizer still shows room for improvement. This, however, consists mostly of implementing standard compiler-construction techniques, and is planned as the next stage of the project.

The central new component described in this paper is the native-code compiler, whose design relies on three ideas to ease implementation: the VM provides services to the native code for an easy implementation or extension of control-transfer-changing instructions; restarting the VM for relieving the native code of certain aspects of the implementation; and the export of VM functionality as C functions for complicated opcodes at the cost of hand-written glue code. We have shown that these three ideas have enabled implementing a simple native-code compiler with little effort, even though it needs to deal with complicated language constructs.

## References

[1] Stephen Adams, Chris Hanson, and the MIT Scheme Team. *MIT Scheme User's Manual*. Available from `http://www.gnu.org/software/mit-scheme/`.

[2] Ali-Reza Adl-Tabatabai, Michal Cierniak, Guei-Yuan Lueh, Vishesh M. Parikh, and James M. Stichnoth. Fast, effective code generation in a just-in-time Java compiler. In Keith Cooper, editor, *Proceedings of the 1998 Conference on Programming Language Design and Implementation*, pages 280–290, Montreal, QC, Canada, June 1998. ACM Press. Volume 33(5) of SIGPLAN Notices.

[3] Bowen Alpern, Dick Attanasio, John Barton, Michael Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Ton Ngo, Mark Mergen, Vivek Sarkar, Mauricio Serrano, Janice Shepherd, Stephen Smith, V. C. Sreedhar, Harini Srinivasan, and John Whaley. The Jalapeño virtual machine. *IBM Systems Journal, Java Performance Issue*, 39(1), 2000.

[4] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Maluszynski and Martin Wirsing, editors, *Proceedings Programming Language Implementation and Logic Programming '91*, volume 528 of *Lecture Notes in Computer Science*, pages 1–13, Passau, Germany, August 1991. Springer-Verlag.

[5] John Aycock. A brief history of just-in-time. *Computing Surveys*, 35(2):97–113, June 2003.

[6] William D. Clinger. Personal communication, November 2006.

[7] William D. Clinger. Rapid case dispatch in scheme. In Robby Findler, editor, *Proceedings of the 2006 Scheme and Functional Programming Workshop*, pages 63–69, Portland, September 2006. University of Chicago TR-2006-06.

[8] William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label or a simple optimizing compiler for scheme. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, pages 128–139, Orlando, FL, USA, June 1994. ACM Press.

[9] William D. Clinger, Anne H. Hartheimer, and Eric M. Ost. Implementation strategies for first-class continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, April 1999.

[10] Timothy Cramer, Richard Friedman, Terrence Miller, David Seberger, Robert Wilson, and Mario Wolczko. Compiling Java just in time. *IEEE Micro*, 17(3):36–43, 1997.

[11] R. Kent Dybvig. The development of Chez Scheme. In Julia Lawall, editor, *Proceedings International Conference on Functional Programming 2006*, pages 1–12, Portland, Oregon, USA, September 2006. ACM Press, New York.

[12] R. Kent Dybvig, David Eby, and Carl Bruggeman. Don't stop the BIBOP: Flexible and efficient storage management for dynamically-typed languages. Technical Report Technical Report #400, Indiana University Computer Science Department, Bloomington, Indiana, March 1994.

[13] Matthew Flatt. *PLT MzScheme: Language Manual*. Rice University, University of Utah, July 2006. Version 352.

[14] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems*. MIT Press, Cambridge, MA, 1985.

[15] Dan Ingalls, Ted Kaehler, John Maloney, Scott Wallace, and Alan Kay. Back to the future: the story of Squeak, a practical Smalltalk written in itself. In *Proceedings of the 12th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications*, volume 32, 10 of *SIGPLAN Notices 32(10)*, pages 318–326, New York, October 1997. ACM Press.

[16] Richard Kelsey, William Clinger, and Jonathan Rees. Revised$^5$ report on the algorithmic language Scheme. *Higher-Order and Symbolic*

*Computation*, 11(1):7–105, 1998.

[17] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Proc. 16th Annual ACM Symposium on Principles of Programming Languages*, pages 281–292, Austin, Texas, USA, January 1989. ACM Press.

[18] Richard A. Kelsey and Jonathan A. Rees. A tractable Scheme implementation. *Lisp and Symbolic Computation*, 7(4):315–335, 1995.

[19] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. In Kathryn S. McKinley, editor, *20 Years of the ACM SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection*, pages 175–191. ACM, April 2004. SIGPLAN Notices 39(4).

[20] Robert A. MacLachlan. The python compiler for cmu common lisp. In *Proceedings 1992 ACM Conference on Lisp and Functional Programming*, pages 235–246, San Francisco, California, USA, June 1992. ACM Press.

[21] Michael Paleczny, Christopher Vick, and Cliff Click. The Java Hotspot$^{TM}$ server compiler. In *USENIX Java Virtual Machine Research and Technology Symposium*, April 2001.

[22] Jonathan A. Rees and Norman I. Adams IV. T: a dialect of lisp or lambda: The ultimate software tool. In *ACM Conference on Lisp and Functional Programming*, pages 114–122, Pittsburgh, Pennsylvania, 1982. ACM Press.

[23] Michael Sperber and Peter Thiemann. Two for the price of one: Composing partial evaluation and compilation. In *Proceedings of the 1997 Conference on Programming Language Design and Implementation*, pages 215–225, Las Vegas, NV, USA, June 1997. ACM Press.

[24] T. Suganuma, T. Ogasawara, K. Kawachiya, M. Takeuchi, K. Ishizaki, A. Koseki, T. Inagaki, T. Yasue, M. Kawahito, T. Onodera, H. Komatsu, and T. Nakatani. Evolution of a Java just-in-time compiler for IA-32 platforms. *IBM Journal of Research and Development*, 48(5/6):767–795, 2004.