Proceedings of the 2013 Federated Conference on
Computer Science and Information Systems pp. 1499–1506

Incremental JIT Compiler for Implicitly Parallel Functional Language

Petr Krajča

Dept. Computer Science, Palacky University, Olomouc
17. listopadu 12, CZ–77146 Olomouc, Czech Republic
petr.krajca@upol.cz

***Abstract*—We present a novel method for automatic parallelization of functional programs which combines interpretation and just-in-time compilation. We propose an execution model for a Lisp-based programming language which involves a runtime environment which is able to identify portions of code worth running in parallel and is able to spawn new threads of execution. Furthermore, in order to achieve better performance, runtime environment dynamically identifies expressions worth compiling and compiles them into a native code.**

I. Introduction

With advent of multi-core processor we are experiencing growing demand for programs which are able to utilize multiple processor cores to increase their efficiency. This trend is likely to continue as the hardware manufacturers have switched their focus from increasing performance by scaling clock speed to developing processors that are able to process multiple tasks simultaneously. Traditionally, programmer has to explicitly define which parts of a program will run in parallel and eventually has to use special means, e.g., locks, semaphores, critical sections, to synchronize parallel branches of execution in order to ensure correct outputs of the program. Currently, this approach to parallel programming is the prevailing one, on the other hand, it is also a very demanding approach from the programmer's point of view.

Another approach, we call it *implicit parallelism*, is to provide a language which is used the same way as any sequential language but its execution can be implicitly parallelized by a compiler or an interpreter of the language. Basically, compiler or runtime environment identifies portions of code which can be run in parallel and takes care of their proper synchronization, and thus, program returns always the same results, no matter which parts of the program were executed simultaneously. If this is achieved, the programmer can completely forget about the issues with parallel execution which is the desirable effect. Nowadays, implicit parallelism is relatively rare and has limited use.

In [14] we outline an evaluation model of the Lisp-based language called Schemik. The execution model of the language is based on a pushdown automaton which allows to describe operational semantics of the language in terms of transitions of the automaton. This model allows for further extensions. The most important extension detects expressions which are worth evaluating in a separate thread of execution and spawns new threads computing values of expressions in parallel. The model has shown its practical merits. (1) Implementation of the model shows that programs can be implicitly parallelized. (2) Using the automaton it can be proved that program returns always the same results no matter which parts of the program run in parallel. (3) One can incorporate software transactional memory to isolate side-effects performed in each thread of execution, hence efficiently parallelize even programs with side effects as discussed in [15]. On the other hand, the model itself is proposed for an interpreted language and this brings significant overhead. This paper presents a new extension for the evaluation model proposed in [14] which incorporates methods of just-in-time (JIT) compilation and preserves all mentioned features of the execution model, especially the ability to run programs in parallel without the need to use dedicated language constructs.

The paper is organized as follows. First, we provide a brief description of the language and of the formal model of the execution, including the methods which are used to automatically parallelize programs. Afterwards, we describe new extension which adds support for JIT compilation. The paper is concluded with a section focusing on implementation details and experimental evaluation.

Supported by grant no. 202/12/P167 of the Czech Science Foundation.

II. Interpreter

A. Evaluation Model: An Overview

In this section, we briefly describe a sequential evaluator which is essential for understanding our approach to parallel computation. We assume that readers are acquainted with some dialect of Lisp or Scheme, and therefore, in the following text we tacitly use the usual terminology which can be found in [4], [9], [12].

Our stack-based evaluator is a deterministic pushdown automaton with two stacks—*execution stack* (denoted E) containing *stack operations* controlling the evaluation, and *result stack* (denoted R) containing Schemik objects playing the roles of operands and intermediate results of operations.

The input (first expression to be evaluated) is encoded on the execution stack and the output (result of evaluation) is pushed on the result stack at the end of the computation. The content of both stacks is what completely describes the current state of the evaluator. Therefore, a setting of the stacks shall be called

978-1-4673-4471-5/$25.00

a *configuration* of the evaluator. The result stack contains first-class objects of Schemik in their internal representation which are basically the same objects as in Scheme [12] (i.e., numbers, symbols, functions, special operators, etc). The execution stack can be seen as a stack for pending operations. Unlike the result stack, it contains stack operations which are not Schemik objects. The stack operations are represented by tuples of the form $\langle$operation-name, $arg, \mathcal{E}$, flag$\rangle$ where operation-name is a name for a step of evaluation (e.g., EVAL, FEVAL, FUNCALL, IF, INSPECT, etc.); *arg* is an object representing argument for stack operation; $\mathcal{E}$ is an environment (i.e., a table describing bindings of lexical variables) associated to the stack operation; flag is an indicator of the tail-recursion optimization [12]. For brevity, attributes *arg*, $\mathcal{E}$, and flag can be omitted from descriptions of a stack operations if they are not used in that particular operation.

During the computation, the automaton changes its configuration. A change from one configuration to another will be called a *transition*. Each transition is determined by stack operation which resides on the top of the execution stack. The computation halts if the execution stack is empty and the object on the top of the result stack is a *result* of the computation. Each transition of the automaton may be depicted by two pairs of stacks—configuration *before* and *after* the transition—and it may be written as follows:

E: *stack with operations before transition* ⟧
R: *stack with results before transition* ⟧
E: *stack with operations after transition* ⟧
R: *stack with results after transition* ⟧

The first pair of stacks represents the configuration *before* the transition. The second pair of stacks, drawn below the first one, represents the configuration *after* the transition. Bottom of all stacks is on the right and is denoted by symbol ⟧.

The *start configuration* of the automaton contains a single stack operation $\langle$EVAL, $expr, \mathcal{E}_t$, N$\rangle$ on the execution stack and an empty result stack, meaning that symbolic expression $expr$ will be evaluated in the top-level environment $\mathcal{E}_t$ (environment containing initial bindings of lexical variables), N says that *expr* does not appear in a tail position, see [12].

The operation $\langle$EVAL, $object, \mathcal{E}$, f$\rangle$ initiates evaluation of $object$ (e.g., a symbolic expression) in environment $\mathcal{E}$. If this operation is on the top of the execution stack, an appropriate transition is made depending on the type of *object*. Three situations may occur: (i) *object* is a self-evaluating object (i.e., neither a symbol nor a non-empty list), in which case *object* is pushed on the result stack; (ii) *object* is a symbol (name of a lexical variable), its value in environment $\mathcal{E}$ is pushed on the result stack; (iii) *object* is a list $(head\ arg_1 \cdots arg_n)$ then the top of the execution stack is replaced as follows:

E: $\langle$EVAL, $(head\ arg_1 \cdots arg_n), \mathcal{E}$, f$\rangle \cdots$ ⟧
R: $\cdots$ ⟧
E: $\langle$EVAL, $head, \mathcal{E}$, N$\rangle$,$\langle$INSPECT, $(arg_1 \cdots arg_n), \mathcal{E}$, f$\rangle \cdots$ ⟧
R: $\cdots$ ⟧

The previous transition has prepared the $head$ of the original list for evaluation. The role of INSPECT is to distinguish between different evaluation rules based on the value of the *head*, i.e. the first element in the list. The *head* may evaluate to a function, a special operator, or a macro. For instance, if INSPECT is on the top of the execution stack and the top of the result stack contains a *function*, INSPECT will prepare application of the function, i.e., it will prepare all arguments for evaluation and then it prepares a function call using FUNCALL:

E: $\langle$INSPECT, $(arg_1 \cdots arg_n), \mathcal{E}$, f$\rangle \cdots$ ⟧
R: $function \cdots$ ⟧
E: $\langle$EVAL, $arg_1, \mathcal{E}$, N$\rangle, \ldots, \langle$EVAL, $arg_n, \mathcal{E}$, N$\rangle$,
$\langle$FUNCALL, $n, \mathcal{E}$, f$\rangle \cdots$ ⟧
R: $function \cdots$ ⟧

The application of functions is performed by FUNCALL which is used in the form $\langle$FUNCALL, $n, \mathcal{E}$, f$\rangle$, where n is the number of arguments. Arguments and the function itself are stored on the result stack. In general, $\langle$FUNCALL, $n, \mathcal{E}$, f$\rangle$ takes $n+1$ objects from the result stack where first n objects represent arguments and the last object taken is the desired function. If the function is a primitive function (i.e., a built-in function), the arguments are passed to the function and the result is pushed on the result stack. If the function is a user-defined function, a body of the function is evaluated in a local environment where formal parameters of the function are bound to arguments obtained from the stack. Therefore, if the body of the function is B, the corresponding transition is the following:

E: $\langle$FUNCALL, $n, \mathcal{E}$, f$\rangle \cdots$ ⟧
R: arg_1, $arg_2, \ldots,$ arg_n, *user-defined fn. with body* $B \cdots$ ⟧
E: $\langle$EVAL, $B, \mathcal{E}'$, T$\rangle \cdots$ ⟧
R: $\cdots$ ⟧

Note that $\mathcal{E}'$ denotes the local environment. According to the value of the flag f, $\mathcal{E}'$ is a fresh new environment (if f equals N) or a reused old environment (if f equals T), which corresponds to a proper tail call, see [12]. For each of the operators, INSPECT has a separate rule of evaluation. For instance, for the special operator `if` the INSPECT operation enforces that only one branch of execution is performed as follows.

Typically, the operator `if` is used in the form: `(if` *cond then-branch else-branch*`)`. If *cond* evaluates to anything but `#f`, the special form returns the value of *then-branch*; otherwise the value of *else-branch* is returned or `#void` is returned in case *else-branch* is not present. Since the result is given by the value of *cond*, INSPECT places the two branches on the result stack, evaluates condition with the operation EVAL, and a new stack operation IF is used to select branch for further evaluation according to the value of *cond*:

E: $\langle$INSPECT, $(cond$ *then-branch else-branch*$), \mathcal{E}$, flag$\rangle \cdots$ ⟧
R: *special operator* `if` $\cdots$ ⟧
E: $\langle$EVAL, $cond, \mathcal{E}$, N$\rangle$,$\langle$IF, $\mathcal{E}$, flag$\rangle \cdots$ ⟧
R: *then-branch*, *else-branch* $\cdots$ ⟧

Now it remains to clarify the newly introduced stack operation IF. The stack operation pops three values from the result stack—an object O representing the value of the *cond*, *then-branch*, and *else-branch*. If O is not equal to `#f`, *then-branch* is pushed back to the execution stack for evaluation:

E: $\langle \mathsf{IF}, \mathcal{E}, \mathsf{flag} \rangle \cdots]\!]$
R: `#f`, *then-branch*, *else-branch* $\cdots]\!]$
E: $\langle \mathsf{EVAL}, \textit{else-branch}, \mathcal{E}, \mathsf{flag} \rangle \cdots]\!]$
R: $\cdots]\!]$

Otherwise, *else-branch* is pushed back to the execution stack for evaluation.

B. Parallel Evaluation Model

In this section, we introduce the implicit parallel evaluation model which is an extension of the evaluation model proposed in the previous section. We outline only the parallel evaluator of a purely functional subset of the language, i.e., subset of Schemik without assignment operations and mutators of pairs (`set!`, `set-car!`, `set-cdr!`), I/O functions (`display`, etc.), and continuations (`call/cc`). Description of the language in its full extent can be found in [16].

Our parallel evaluator of purely functional subset of the language is based on the idea that each operation EVAL may be processed in an independent evaluator as far as the resulting value is dependent only on given arguments and environment. We introduce a new stack operation FEVAL (abbreviation for "fork eval") representing evaluation in an *independent evaluator*, i.e., a thread. Basically, all the independent evaluators form a tree structure with a single (main) evaluator at the top of the hierarchy. Each evaluator has its own independent pair of stacks and environments are shared among all evaluators. We can share environments without any inconsistency issues because we are considering programs without side effects only. This means, once an environment is established, it does not change bindings of symbols (lexical variables).

C. Stack Operation FEVAL

Operation FEVAL is similar to EVAL; it prepares an expression for evaluation in an independent evaluator, i.e., using a new pair of stacks. Operation FEVAL is specific in that it is not a result of any transition caused by another operation. The appearance of FEVAL on any execution stack is caused by external entity called *scheduler* which converts a suitable EVAL operation on an execution stacks to the FEVAL operation and creates a new evaluator containing the original EVAL on top of its execution stack. This way a parallel branch of the evaluation is created. The transition created by scheduler may be depicted as follows:

EV_1: E: $\cdots \langle \mathsf{EVAL}, \textit{object}, \mathcal{E}, \mathsf{f} \rangle \cdots]\!]$
R: $\cdots]\!]$
$\vdots$
EV_1: E: $\cdots \langle \mathsf{FEVAL}, \langle EV_2, \textit{object} \rangle, \mathcal{E}, \mathsf{f} \rangle \cdots]\!]$
R: $\cdots]\!]$
EV_2: E: $\langle \mathsf{EVAL}, \textit{object}, \mathcal{E}, \mathsf{N} \rangle]\!]$
R: $]\!]$

An invocation of FEVAL represents merging of two branches of the evaluation. If $\langle \mathsf{FEVAL}, \langle EV_i, \textit{object} \rangle, \mathcal{E}, \mathsf{f} \rangle$ appears on the top of the execution stack, the evaluation in the referred evaluator EV_i is stopped and its stacks are appended to the corresponding stacks of the current evaluator processing the FEVAL operation.

This behavior of FEVAL is correct because the referenced evaluator EV_i has performed right the same transitions that should be otherwise performed by the evaluator containing the FEVAL operation. We can consider the creation of this operation as a request for parallel evaluation to other evaluator and processing of FEVAL operation as a request for obtaining (partial) results from the referenced evaluator.

D. Scheduling Strategies

An important issue in implicit parallel evaluation is how to decide which expression is worth evaluating in a parallel branch. In theory, an arbitrary expression may be parallelized but it is desirable to focus on particular expressions that require larger amounts of transitions.

The decision on which expression will be evaluated in a parallel branch is done by the *scheduler*, associated to each evaluator, regularly seeking through the execution stack for operations EVAL that may be converted into the FEVAL operations. A stack operation $\langle \mathsf{EVAL}, \textit{object}, \mathcal{E}, \mathsf{f} \rangle$ is considered as a good candidate for parallel evaluation if $\textit{object}$ is a list or a list fulfilling further criteria. For instance, $\textit{object}$ should be a list containing other list, or a list with a user-defined function as its first element.

In particular, we use strategy assigning limited number of independent evaluators to candidate expressions which has its origin in the organization of the execution stack. This strategy assumes that more complex expressions are at the bottom of the stack and their evaluation will need more transitions, and thus, their evaluation in a parallel branch will be more beneficial. Therefore, scheduler walks through the execution stack from the bottom and looks for candidate expressions.

III. Just-in-Time Compilation

The ability of the execution model to automatically parallelize programs comes from the fact that one can easily determine the structure of the program execution by inspection of the automaton. Particularly, by inspection of the *execution stack*. However, if the program, or its portion, is compiled, we are losing the ability to analyze the program structure since scheduler is unable to identify expression for parallelization. In order to combine JIT compilation with the ability to parallelize programs, we propose an extension which compiles into a native code only expressions which are insignificant from the point of parallelization, and thus, compiler does not interfere with the mechanism for automatic parallelization. Such typical insignificant expression is an atom (symbol or self-evaluating object) or function call where arguments are atomic expressions. Apparently, their evaluation is very fast and their evaluation in parallel branches would lead to fine-grained parallelism and to a related overhead caused by

scheduling and spawning too many threads of execution. These expressions which are insignificant from the point of view of parallelization and which are, on the other hand, worth compiling shall be called *compilable*. Formally, an expression E is *compilable* if it satisfies one of the following conditions:

(1) Expression E is either atom (i.e., symbol, number, string, etc.),
(2) or E is an expression of a form $(E_1\ E_2\ \ldots E_n)$ where E_1 is a primitive function or a special operator and $E_2, \ldots, E_n$ are all compilable expressions.

Remark 3.1: Notice that the definition of compilable expression is recursive. Later, we use this fact to incrementally compile complex expression using the less complex ones.

Remark 3.2: It might be impossible to decide whether expression is compilable or not, strictly speaking, generally, it is an undecidable problem. Therefore, in a case when we are unable to determine if an expression is compilable, we assume that the expression is not compilable at all. When determining compilability of a non-atomic expression, it is crucial to determine the type of the first expression. For this purpose we use the following approximation. If the first expression E_1 is a symbol defined only in the top-level environment (w.r.t. the environment in which E_1 is evaluated) and if a primitive function or a special operator is bounded to this symbol, we assume that E_1 evaluates to a primitive function or a special operator. Note that this condition can be checked in a constant time and covers among others common function calls where symbol is used to directly represent a primitive function. For instance, `(+ a 1)` is an example of such expression.

A. Compilation: An Overview

The compilation process consists of several independent steps.

Before the program execution starts, all lists in the source code consisting solely of atoms are marked with a flag indicating that the given expression is a good candidate for compilation. For this purpose all source code expressions are equipped with a data structure similar to property-lists known from Common Lisp [9] which contains flags and further information produced by compiler, e.g., native code. Notice that this flag does not mean that expression is compilable, rather it is an indicator that compiler should try to compile this expression.

Every time the operation EVAL is on the top of the execution stack, its argument (evaluated expression) is checked if it has associated native code generated by the compiler, if so, the given code is executed and returned value is pushed on the result stack as if the evaluation was performed in a usual way. Otherwise, if the given expression is marked as a good candidate for a compilation, it is inserted into a queue of expressions waiting for compilation. In any case, if the expression has not attached any native code, it is evaluated in a standard way as described in Section II.

All expressions waiting for compilation in a queue are processed one by one by a compiler. For each expression, compiler checks whether is compilable, and if not, flag representing that an expression is a good candidate for compilation is removed. Otherwise, expression is compiled and the results of the compilation process (native code and intermediate representation of the code) are attached to the expression. Furthermore, expression containing this expression is marked as a good candidate for compilation.

Remark 3.3: Marking a parent expression as a candidate expression for compilation is a necessary step allowing us to incrementally compile complex expressions. For example, initially entire expression `(+ a (+ b 1))` is not marked as a good candidate for compilation since the second argument is not an atom and one can not decide if `(+ b 1)` is compilable. However, subexpression `(+ b 1)` is marked as a good candidate. If this subexpression is compiled, an entire expression `(+ a (+ b 1))` becomes also a good candidate and may be compiled into a native code. This corresponds with the recursive definition of *compilable* expression.

The key part of the compilation process is a transformation of a given expression into a native code, this transformation and related implementation aspects are discussed in the next section.

B. Compilation: Code Generation

The main task of the compiler is to take an expression and transform it into a native code which under the same conditions evaluates to same value as it would be evaluated usually. For instance, having an expression `(+ a 1)` we need to generate function with a prototype like this:

```
value *plus_a_1(environment *)
```

which gets value of `a` in a given environment and returns its value increased by one.

The process of expression compilation we propose can be divided into three steps. At first, expression is compiled into a high-level intermediate representation (HIR), this representation by its nature is very close to three-address code used by many compilers. In the next step optimization techniques are applied. Afterwards, HIR is transformed into a native code. Optionally, this step may consist of additional transformation. For instance, we transform HIR into a corresponding low-level intermediate representation (LIR) which is used to perform low-level optimizations, and after that, the native code is generated.

The high-level intermediate representation we propose consists of operations having up to three arguments. These arguments may be constants (regular Schemik objects), registers playing the role of (temporary) variables, or lists of operations of HIR playing the role of code blocks. We assume that registers are enumerable and shall be denoted R_i where i is an integer. List of essential operations used by HIR along with their semantics is presented in Table I.

The recursive way we define compilable expressions has been reflected in a way we compile expressions into a HIR. Entire process of transformation of the expression to HIR is described by the recursive procedure COMPILEHIR procedure

TABLE I
LIST OF OPERATIONS USED IN HIGH-LEVEL INTERMEDIATE REPRESENTATION

operation	meaning
`set` R_i, *source*	assigns value of *source* to the register R_i; *source* may be either a constant or register
`eval-symbol` R_i, S	evaluates symbol S w.r.t. current environment and stores the result into the register R_i
`prepare` *count*	an auxiliary operation introducing each function call, its purpose is to declare number of arguments
`putarg` i, *source*	passes value of *source* as the i-th argument of the function; *source* may be either a constant or register
`funcall` R_i, *fun*	calls function *fun* and stores the returned value into the register R_i
`add` R_i, R_j, *value*	adds R_j and *value*, which may be a constant or register, and stores the result into R_i; note that there are also operations for subtraction, multiplication, comparisons, etc.
`car` R_i, R_j	extracts the *car*-part of a dotted pair stored in the register R_j and stores the result into the register R_i
`cdr` R_i, R_j	analogous operation to `car` extracting the *cdr*-part of a dotted pair
`if` R_i, BRANCH$_1$, BRANCH$_2$	if value in the register R_i is other then `#f`, code in BRANCH-1 is performed, otherwise, BRANCH-2 is executed; note that BRANCH-1, BRANCH-2 represents an entire list of operations in HIR, i.e., correspond to code blocks
`rslt-push` R_i	pushes value in the register R_i on top of the result stack
`exct-push` *expression*	pushes value of *expression* on the execution stack; *expression* may be either a constant or register

which is presented in Algorithm 1. This procedure has two arguments—the given expression E and register number $base$. This number identifies the first register that can be used to compute the value of the expression E. The crucial feature of this procedure is that COMPILEHIR does not generate code directly, but returns a procedure which generates the final HIR for the expression. This procedure has one argument i and allows to shift used registers by offset i. In other words, COMPILEHIR does not generate final code, but rather a template of the code. This behavior is illustrated in Figure 1 (top) containing the result of COMPILEHIR procedure for expression `(foo a 1)` and argument $base$ equal to 10. As one can see this "template" is very close to three address code and the main difference is that registers does not have fixed numbers and may be shifted by offset i. Nonetheless, without loss of generality we may apply optimization techniques proposed for three-address code, e.g., constant and copy propagation, or dead-code elimination.

We now turn our attention to the HIR(i) procedure which is a result of the COMPILEHIR(E, $base$) procedure, as described in Algorithm 1. We assume that the generated code satisfies

Algorithm 1 Procedure COMPILEHIR(E, $base$)

```
1:  return procedure HIR(i) such that:
2:  if E is a constant (e.g., number) then
3:      emit operation set R_{base+i}, E
4:  if E is a symbol then
5:      emit operation eval-symbol R_{base+i}, E
6:  if E has attached HIR code then
7:      invoke HIR(base + i)
8:  if E is an expression (fun E_2 E_3 ... E_n) where fun
    is a primitive function then
9:      for all E_j where j ∈ {2, ..., n} do
10:         invoke COMPILEHIR(E_j, base + i + j − 1)
11:     if operation exct-push was emitted then
12:         abort compilation
13:     if fun is primitive function + then
14:         invoke COMPILEADDITION(base + i, n)
15:     else if fun is primitive function car then
16:         emit operation car R_{base+i}, R_{base+i+1}
17:     else
18:         invoke COMPILEFUNCALL(base + i, n, fun)
19: if E is expression (if E_cond E_1 E_2) and E_cond is
    compilable then
20:     invoke COMPILEIF(base + i, E_cond, E_1, E_2)
21: if E is quotation (quote val) then
22:     emit operation set R_{base+i}, val
23: otherwise abort compilation
```

the following conditions:

(i) The result of the computation is stored in the register R_{base+i} or generated code modifies content of the evaluator's stacks directly and in that case no value is returned.
(ii) No register with an index lesser than $base + i$ will be used to compute the result.

If the expression is an atom (see Algorithm 1, lines 2–5), procedure HIR generates operation which gets the value of the atom and stores the result into the register R_{base+i}. In case we encounter an expression which has attached code in HIR, see lines 6–7, we use the fact that intermediate code is already available and we use it to compute value of the expression. Notice that HIR is in fact a procedure and is invoked with an argument $base+i$. This ensures that once generated HIR code may use different set of registers which does not interfere with the registers already used. Furthermore, from the assumption (i) follows that the result, if returned, will be in the register R_{base+i}.

Code generation for non-atomic expressions consists of multiple sub-actions, see lines 8–22. Initially, the value of the first element is determined, i.e., the called function is determined. Subsequently, code that computes values of all arguments is generated, assuming that all arguments are evaluated in the left-to-right order and arguments are stored in registers $R_{base+i+1}$, $R_{base+i+2}$, …, $R_{base+i+n}$. To generate code evaluating arguments, we recursively invoke the COM-

Procedure HIR(i):
emit operations:
`eval-symbol` R_{11+i}, a
`set` R_{12+i}, 1
`prepare` 2
`putarg` 1, R_{11+i}
`putarg` 2, R_{12+i}
`funcall` R_{10+i}, *foo*

Procedure HIR(i):
emit operations:
`eval-symbol` R_{11+i}, a
`if` R_{11+i}, {
`set` R_{11+i}, 0
`rslt-push` R_{11+i}
}, {
`exct-push (foo b)`
}

Fig. 1. HIR of `(foo a 1)` (top) and HIR of `(if a 0 (foo b))` (bottom)

PILEHIR procedure at lines 9–10. Due to the assumption (ii), it is guaranteed that the registers used to compute arguments will not overlap in an undesirable way. This means, values of already computed arguments which are stored in the registers will not be overwritten during the computation of remaining arguments. Note that if any argument is compiled expression which does not return a value and rather directly manipulates with stacks, it is a sign that we can not obtain a value of this argument and entire expression is not compilable and compilation process has to be aborted, see lines 11–12. Situation when the generated code directly tempers with the execution stack is discussed later.

If the called function is a primitive function and if the compiler has mean to compile this function into a native code, appropriate code is generated. This is the case of frequently used functions, for instance, function for addition, subtraction, `car`, `cdr`, etc. Procedure presented in Algorithm 1 shows how this rule applies for addition and function `car`, see lines 13–16 and related Algorithm 2. Otherwise, code for generic function call is generated, see line 18 in Algorithm 1 and Algorithm 3.

Remark 3.4: In case of frequently called functions (e.g., addition) it may be efficient to generate code directly instead of performing generic function call, since calling a function in native code has overhead related to argument passing.

Compilation of special operators requires different approach. In this paper we discuss only two operators—`if` and `quote` since our experiments have shown, these operators affect the program performance the most. Compilation of the `(quote val)` expression is straightforward, it suffices to assign value of `val` to the register R_{base+i}. In case of the `if` operator is situation much more complex.

Let us consider the following two snippets of source code `(if (> a 0) 1 (+ a 1))` and `(if (> a 0) 1 (foo a))` and let us assume that `(> a 0)`, `(+ a 1)` are compilable expressions and

Algorithm 2 Procedure COMPILEADDITION(i, n)

1: **emit** operations:
`set` R_i, 0
`add` R_i, R_i, R_{i+1}
...
`add` R_i, R_i, R_{i+n}

`(foo a)` is not compilable. In the first case we have an expression which can be safely compiled because all subexpressions are compilable. In the second case we have limited information on `(foo a)` and we have to assume that this expression may perform any sequence of operations, and thus, is significant from the parallelization point of view. In conclusion, we may assume that only `(> a 0)` is compilable. Unfortunately, the second type of conditions appears in real-world programs more often than the first one, hence we have to take this fact into account.

The compilation process of conditional expression is presented in the COMPILEIF procedure (described in Algorithm 4) which is invoked from COMPILEHIR. While compiling conditional expression, we always assume that the condition is compilable expression and its HIR is available, see lines 1–4 in Algorithm 4. For expressions representing branches two situations may occur. (1) Both branches are compilable and HIR is available, (2) or at least one expression is not compilable or HIR is not available for this expression. In the first case, we generate a code which computes value of the condition and if the value is true or false, it computes value of the first or the second branch, respectively, and stores the result into a given register, see lines 5–10 and 15 in Algorithm 4. Notice that this case corresponds to the first snippet of the code discussed in the previous paragraph. In the second case, which corresponds to the second code snippet from the previous paragraph, we are unable to determine the value of the expression entirely and at some point we have to pass control back to the evaluator. Basically, generated code has to compute value of the condition and accordingly perform one branch of the execution. Code generated for each branch depends on the nature of the expression in a given branch. If HIR of the expression is available, this code is returned along with the operation `rslt-push` which pushes result on the *result stack* and passes execution back to the evaluator, see lines 11–14 in Algorithm 4. Otherwise, operation `exct-push` is performed, this operation pushes given expression on the execution stack, in other words, it evaluates expression by means of the evaluator. An illustrative example of compiled conditional expression `(if a 0 (foo b))`, which is generated by the COMPILEIF procedure, is presented in Figure 1 (bottom).

C. Native Code Generation

Using the COMPILEHIR procedure we have obtained high-level intermediate representation of the expression. This representation by its nature allows multiple optimizations, especially, those based on data-flow analysis. Since the HIR is very

Algorithm 3 Procedure COMPILEFUNCALL(i, n, *fun*)

emit operations:
`prepare` n
`putarg` 1, R_{i+1}
`putarg` 2, R_{i+2}
...
`putarg` n, R_{i+n}
`funcall` R_i, *fun*

Algorithm 4 Procedure COMPILEIF(i, E_{cond}, E_1, E_2)

if E_{cond} has attached HIR code without the `exct-push` operation **then**
    invoke HIR(i)
else
    abort compilation
for all E_j where $j \in \{1, 2\}$ **do**
    // create code block BRANCH$_j$ *such that:*
    if E_j has attached HIR code **then**
        BRANCH$_j$ $\leftarrow$ HIR(i)
    else
        BRANCH$_j$ $\leftarrow$ operation `exct-push` E_j
if E_1 has attached HIR code **and** BRANCH$_2$ contains `exct-push` **then**
    append to BRANCH$_1$ operation `rslt-push` R_1.
if E_2 has attached HIR code **and** BRANCH$_1$ contains `exct-push` **then**
    append to BRANCH$_2$ operation `rslt-push` R_2
emit operation `if` R_i, BRANCH$_1$, BRANCH$_2$

close to three-address code, we may directly apply optimization techniques. Namely we use copy propagation, constant propagation, constant folding, and dead code elimination. All these techniques significantly simplify the resulting code. These techniques are well-known and thoroughly studied, for instance, in [18] or [2], therefore we omit discussion on this topic and refer readers to these books.

Apparently, each operation of the HIR may be transformed into an assembly language. Usually, it requires less than ten instructions of assembly language to express these operations, because operations of HIR are in fact very simple. However, in order to increase portability of the compiler, our implementation converts HIR into a low-level intermediate representation (LIR) which is afterwards compiled to a native code, by a library. Our implementation is based on the MyJIT library [1], therefore our low-level intermediate representation is equivalent to the instruction set of this library. This library generates machine code for numeric operations, conditional and unconditional jumps, function calls, memory access operations, and takes care of register allocation. Since the LIR is tightly coupled with implementation aspects of the interpreter/compiler, we do not describe transformation from the HIR to the LIR and code generation in this paper. Nonetheless, it could be possible to use any other similar library to generate native code, for instance, LLVM.

Major part of our JIT compiler is written in Schemik itself, i.e., the JIT compiler is self-hosted and its code is a part of the standard library and all intermediate representations are objects (lists) in Schemik language as well. This way we benefit from the expressiveness of the Schemik language, especially, from the ability to conveniently transform one list of values to another. Every time the evaluator encounters compilable expression, it invokes the `jit:compile` function which is an ordinary function written in Schemik taking care of the transformation of an expression into the HIR and, subsequently, into a native code. For this purpose, Schemik has bindings to the MyJIT library. Function `jit:compile` returns representation of the expression in the HIR and a native code and an evaluator assigns these results to a given expression. The `jit:compile` function is always evaluated in a separate thread, however, it is evaluated with a regular evaluator as discussed in Sections II and III, hence benefits from the parallel evaluation of expressions and even from the JIT compilation of expressions. Since the compilation of the user supplied source code is mixed together with the compilation of the compiler itself, it usually takes some minimal amount of time for compiler to take effect, because compiler tends to compile itself first. However, if compiled version of an expression is not available, regular evaluation process is used, hence compilation has minimal impact on the execution time, if compared with the regular evaluator.

IV. EXPERIMENTS

In order to evaluate impact of the JIT compiler we performed set of experiments on a computer equipped with two Intel Xeons E5642 (eight CPU-cores in total) using standard benchmarks. Results of these experiments are presented in Table II containing times needed to compute given task for various configurations of the interpreter. First two columns represent results for an interpreter/compiler running only with one thread. The next two columns represent results when interpreter/compiler was allowed to use up to eight threads to compute results. The last column shows the results for the Guile Scheme (1.8.8). Experiments shows typical performance improvement by factor 2 or 3 depending on the type of benchmark and data size. It also shows that the results are fully comparable with Guile.

Important feature of parallel systems is their scalability. Our experiments show that the compiler slightly decreases scalability, see Figure 2. However, this is mostly only an effect of the Amdahl's law. Nevertheless, this shortcoming is adequately compensated since the compiler significantly improves performance, as it is apparent from Table II.

V. RELATED WORKS

Basically, two approaches to impose implicit parallelism into programming languages can be distinguished. The first is static code analysis which hard-wires parallel execution into a program during the compilation. This technique proved to be useful and has become part of the commercial compilers.

TABLE II
RESULTS OF BENCHMARKS (IN SECONDS) FOR VARIOUS CONFIGURATION OF THE EVALUATOR

	1 thread		8 threads		
	No JIT	JIT	No JIT	JIT	Guile
bubblesort	5.77	3.27	5.81	3.29	1.35
combinations	2.74	1.62	1.33	0.94	1.84
cpstak	8.77	5.19	2.08	1.40	2.79
fib30	1.23	0.49	0.43	0.30	0.31
fib33	5.24	2.03	1.69	0.87	1.27
fib35	13.62	5.19	4.49	2.42	3.32
mazefun	7.62	3.55	1.69	1.15	2.24
mergesort	6.53	3.70	2.99	1.69	0.14
nqueens	3.68	1.80	1.30	1.08	0.64
powerset	2.41	1.53	1.78	0.95	1.97
primes	8.65	3.63	4.11	3.48	1.86
quicksort	6.33	2.24	3.79	1.60	3.70
sum	8.08	2.78	3.33	2.84	2.31
tak	5.72	1.49	1.31	0.81	1.73

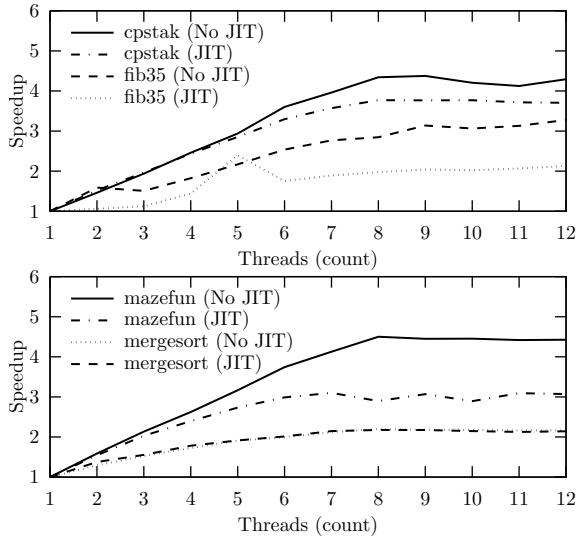

Fig. 2. Scalability of standard benchmarks

However, static code analysis usually focus only on loop parallelization, see [5], [13], what limits its use.

The second approach utilizes run-time analysis and dynamic parallelization. From this category most influential programming languages are Fortress [3] and Concurrent Haskell [19], [6]. However their execution model is tied to specifics of given languages and therefore it is hardly possible to transfer their model into other languages, e.g., into Scheme. Idea to parallelize Lisp-programs is not novel and many approaches to parallel execution of programs in Lisp [17] and Scheme [8], [10] appeared. Nevertheless, majority of this approaches were focusing on explicit parallelism, survey of these approaches can be found in [11]. Relatively, new approach to automatic parallelization of Scheme programs is proposed in [7] which is an interpretation model based on a speculative execution model. To the best our knowledge there is no other execution model like ours which combines run-time analysis and dynamic parallelization along with compilation.

VI. CONCLUSIONS

We have proposed an extension of the execution model for an implicitly parallel programing language. Our extension introduces means of JIT compilation into the execution model which was originally intended for interpreted programming language. Our experiments show that JIT compiler can efficiently decrease execution time of programs while preserving all features of the underlying model language, especially the ability to automatically parallelize programs. In future we intend to focus on further optimizations which are able to decrease program execution time even more, e.g., function inlining or specialization.

REFERENCES

[1] Myjit. http://myjit.sourceforge.net.
[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.
[3] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification Version 1.0, March 2008.
[4] John Allen. *Anatomy of LISP*. McGraw-Hill, Inc., New York, NY, USA, 1978.
[5] Pierre Boulet, Alain Darte, Georges-Andr Silber, and Frdric Vivien. Loop parallelization algorithms: From parallelism extraction to code generation. *Parallel Computing*, pages 421–444, 1998.
[6] Tim Harris, Simon Marlow, Simon L. Peyton Jones, and Maurice Herlihy. Composable memory transactions. In Keshav Pingali, Katherine A. Yelick, and Andrew S. Grimshaw, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (10th PPOPP'2005), ACM SIGPLAN Notices*, pages 48–60, Chicago, IL, USA, June 2005.
[7] Charlotte Herzeel and Pascal Costanza. Dynamic parallelization of recursive code: part 1: managing control flow interactions with the continuator. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 377–396, New York, NY, USA, 2010. ACM.
[8] Guy L. Steele Jr. Lambda: The Ultimate Declarative. AI Memo 379, MIT AI laboratory, November 1976.
[9] Guy L. Steele Jr. *Common LISP the Language*. Digital Press, 2nd edition, 1990.
[10] Guy L. Steele Jr. and Gerald J. Sussman. Lambda: The Ultimate Imperative. AI Memo 353, MIT AI laboratory, March 1976.
[11] Robert H. Halstead Jr. New Ideas in Parallel Lisp: Language Design, Implementation, and Programming Tools. In Takayasu Ito and Robert H. Halstead Jr., editors, *Workshop on Parallel Lisp*, volume 441 of *Lecture Notes in Computer Science*, pages 2–57. Springer, 1989.
[12] Richard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised^5 Report on the Algorithmic Language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.
[13] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.
[14] Petr Krajca and Vilém Vychodil. Data parallel dialect of scheme: outline of the formal model, implementation, performance. In Sung Y. Shin and Sascha Ossowski, editors, *SAC*, pages 1938–1939. ACM, 2009.
[15] Petr Krajca and Vilém Vychodil. Software transactional memory for implicitly parallel functional language. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal, and Chih-Cheng Hung, editors, *SAC*, pages 2123–2130. ACM, 2010.
[16] Petr Krajca and Vilém Vychodil. Stack-based model of implicit parallel execution of functional programs. In preparation.
[17] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.
[18] Steven S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.
[19] Simon Peyton Jones, Andrew Gordon, and Sigbjorn Finne. Concurrent haskell. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, pages 295–308, New York, NY, USA, 1996. ACM.