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 the 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.

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...
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 stacks operations which are not Schemik objects. The stack operations are represented by tuples of the form (operation-name, arg, E, flag) 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; 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, 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 (EVAL, exp, E_0, N) on the execution stack and an empty result stack, meaning that symbolic expression exp will be evaluated in the top-level environment E_0 (environment containing initial bindings of lexical variables), N says that exp does not appear in a tail position, see [12].

The operation (EVAL, object, E, f) initiates evaluation of object (e.g., a symbolic expression) in environment 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 E is pushed on the result stack; (iii) object is a list (head, arg_1, · · · , arg_n) then the top of the execution stack is replaced as follows:

E: (EVAL, head, arg_1, · · · , arg_n), E, f)]
R: · · · ]
E: (EVAL, head, E, N), (INSPECT, arg_1, · · · , arg_n), E, f)]
R: · · · ]

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: (INSPECT, (arg_1, · · · , arg_n), E, f)]
R: function · · · ]
E: (EVAL, arg_1, E, N), · · · , (EVAL, arg_n, E, N),
(fUNCALL, n, E, f)]
R: function · · · ]

The application of functions is performed by FUNCALL which is used in the form (FUNCALL, n, E, f), where n is the number of arguments. Arguments and the function itself are stored on the result stack. In general, (FUNCALL, n, E, f) 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: (FUNCALL, n, E, f)]
R: arg_1, arg_2, · · · , arg_n, user-defined fn. with body B · · · ]
E: (EVAL, B, E', T)]
R: · · · ]

Note that E' denotes the local environment. According to the value of the flag f, 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: (INSPECT, (cond, then-branch, else-branch), E, flag)]
R: special operator · · · ]
E: (EVAL, cond, E, N), (IF, E, flag)]
R: then-branch, else-branch · · · ]
Now it remains to clarify the newly introduced stack operation \texttt{IF}. The stack operation pops three values from the result stack—an object $O$ representing the value of the \texttt{cond}, then-branch, and else-branch. If $O$ is not equal to \#f, then-branch is pushed back to the execution stack for evaluation:

\[
\text{E: } [\text{IF}, \text{flag}] \ldots ] \\
\text{R: } [\#f, \text{then-branch}, \text{else-branch}] \ldots ] \\
\text{E: } [\text{EVAL}, \text{else-branch}, \text{flag}] \ldots ] \\
\text{R: } [\ldots ]
\]

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

\section*{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 \texttt{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 \texttt{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).

\section*{C. Stack Operation \texttt{FEVAL}}

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

\begin{align*}
\text{EV}_1: & \quad \text{E: } [\ldots [\text{EVAL}, \text{object}, \text{flag}] \ldots ] \\
& \quad \text{R: } [\ldots ] \\
& \quad \ldots \\
\text{EV}_1: & \quad \text{E: } [\ldots [\text{FEVAL}, \{\text{EV}_2, \text{object}\}, \text{flag}] \ldots ] \\
& \quad \text{R: } [\ldots ] \\
\text{EV}_2: & \quad \text{E: } [\text{EVAL}, \text{object}, \text{flag}] \\
& \quad \text{R: } [\ldots ]
\end{align*}

An invocation of \texttt{FEVAL} represents merging of two branches of the evaluation. If \{\texttt{FEVAL}, \{\texttt{EV}_1, \text{object}\}, \text{flag}\} appears on the top of the execution stack, the value in the referred evaluator \texttt{EV}_1 is stopped and its stacks are appended to the corresponding stacks of the current evaluator processing the \texttt{FEVAL} operation.

This behavior of \texttt{FEVAL} is correct because the referenced evaluator \texttt{EV}_1 has performed right the same transitions that should be otherwise performed by the evaluator containing the \texttt{FEVAL} operation. We can consider the creation of this operation as a request for parallel evaluation to other evaluator and processing of \texttt{FEVAL} operation as a request for obtaining (partial) results from the referenced evaluator.

\section*{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 \texttt{scheduler}, associated to each evaluator, regularly seeking through the execution stack for operations \texttt{EVAL} that may be converted into the \texttt{FEVAL} operations. A stack operation \{\texttt{EVAL}, \text{object}, \text{flag}\} is considered as a good candidate for parallel evaluation if \text{object} is a list or a list fulfilling further criteria. For instance, \text{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, \texttt{scheduler} walks through the execution stack from the bottom and looks for candidate expressions.

III. \textsc{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 \textit{execution stack}. However, if the program, or its portion, is compiled, we are losing the ability to analyze the program structure since \texttt{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 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:

\[
\text{value *plus_a_1(environment *)}
\]

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 is 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 COMP IL E HIR procedure
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 COMPILIEIR 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, COMPILIEIR does not generate final code, but rather a template of the code. This behavior is illustrated in Figure 1 (top) containing the result of COMPILIEIR 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 COMPILIEIR($E$, $base$) procedure, as described in Algorithm 1. We assume that the generated code satisfies 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 $R_{base + i1}$, $R_{base + i2}$, ..., $R_{base + in}$. To generate code evaluating arguments, we recursively invoke the CO
Algorithm 2 Procedure COMPILEADDITION(i, n)

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

(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 COMPILIEF 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 COMPILIEF 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
Nonetheless, it could be possible to use any other similar from the HIR to the LIR and code generation in this paper.

The interpreter/compiler, we do not describe transformation of the LIR is tightly coupled with implementation aspects of access operations, and takes care of register allocation. Since conditional and unconditional jumps, function calls, memory

This library generates machine code for numeric operations, Amdahl's law. Nevertheless, this shortcoming

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.

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.

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.

Algorithm 3 Procedure COMPILFUNCFUNCALL(i, n, fun)

1: emit operations:
2: prepare n
3: putarg 1, R_{i+1}
4: putarg 2, R_{i+2}
5: ... 
6: putarg n, R_{i+n}
7: funcall R_i, fun

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

1: if E_{cond} has attached HIR code without the exct-push operation then
2: invoke HIR(i)
3: else
4: abort compilation
5: for all E_j where j \in \{1,2\} do
6: // create code block BRANCH_j such that:
7: if E_j has attached HIR code then
8: BRANCH_j <- HIR(i)
9: else
10: BRANCH_j <- operation exct-push E_j
11: if E_1 has attached HIR code and BRANCH_2 contains exct-push then
12: append to BRANCH_1 operation rslt-push R_1.
13: if E_2 has attached HIR code and BRANCH_1 contains exct-push then
14: append to BRANCH_2 operation rslt-push R_2
15: emit operation if R_1, BRANCH_1, BRANCH_2

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.
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


### Table II

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>1 thread</th>
<th>8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No JIT</td>
<td>JI T</td>
</tr>
<tr>
<td>bubblesort</td>
<td>5.77</td>
<td>3.27</td>
</tr>
<tr>
<td>combinations</td>
<td>2.74</td>
<td>1.62</td>
</tr>
<tr>
<td>cpstk</td>
<td>8.77</td>
<td>5.19</td>
</tr>
<tr>
<td>fib30</td>
<td>1.23</td>
<td>0.49</td>
</tr>
<tr>
<td>fib33</td>
<td>5.24</td>
<td>2.03</td>
</tr>
<tr>
<td>fib35</td>
<td>13.62</td>
<td>5.19</td>
</tr>
<tr>
<td>mazefun</td>
<td>7.62</td>
<td>3.55</td>
</tr>
<tr>
<td>mergesort</td>
<td>6.53</td>
<td>3.70</td>
</tr>
<tr>
<td>nqueens</td>
<td>3.68</td>
<td>1.80</td>
</tr>
<tr>
<td>powerset</td>
<td>2.41</td>
<td>1.53</td>
</tr>
<tr>
<td>primes</td>
<td>8.65</td>
<td>3.63</td>
</tr>
<tr>
<td>quicksort</td>
<td>6.33</td>
<td>2.24</td>
</tr>
<tr>
<td>sum</td>
<td>8.08</td>
<td>2.78</td>
</tr>
<tr>
<td>tak</td>
<td>5.72</td>
<td>1.49</td>
</tr>
</tbody>
</table>

### 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