Automatic Parallelization of Side-Effecting Higher-Order Scheme Programs

Jens Nicolay, Coen De Roover, Wolfgang De Meuter, Viviane Jonckers
Software Languages Lab
Vrije Universiteit Brussel
jnicolay,cderoove,wdmeuter,vejoncke@vub.ac.be

***Abstract*—The multi-core revolution heralds a challenging era for software maintainers. Manually parallelizing large sequential code bases is often infeasible. In this paper, we present a program transformation that automatically parallelizes real-life Scheme programs. The transformation has to be instantiated with an interprocedural dependence analysis that exposes parallelization opportunities in a sequential program. To this end, we extended a state-of-the art analysis that copes with higher-order procedures and side effects. Our parallelizing transformation exploits all opportunities for parallelization that are exposed by the dependence analysis. Experiments demonstrate that this brute-force approach realizes scalable speedups in certain benchmarks, while others would benefit from a more selective parallelization.**

***Keywords*-parallelization, static analysis, program transformation**

I. Introduction

Multi-core processors are here and are here to stay for the foreseeable future. This revolution heralds a challenging era for software maintainers. While parallelism is now supported by commodity hardware, it still has to be exploited in software to realize a performance increase. For large sequential code bases, manual parallelization might not prove economically viable at all. Invasive, error-prone and labor-intensive changes can only be justified if they realize the expected speedup. Alternatively, the compiler or interpreter can be tasked with introducing parallelism in the execution of a sequential program. For instance, different loop iterations or recursive procedure applications can be executed in parallel. While such automatic parallelization techniques have realized speedups in high-performance computing, architectural contrasts (e.g., memory models, interconnection bandwidths), radically different problem domains (e.g., back office applications versus particle simulations) and the use of high-level language features (e.g., closures, polymorphism, reflection) encumber transposing them into the multi-core era.

In this paper, we describe a program transformation that automatically parallelizes higher-order side-effecting Scheme programs. The transformation must be instantiated with a sound and sufficiently precise dependence analysis, for dependent expressions cannot safely be evaluated in parallel. We build on the state-of-the-art dependence analysis by Might and Prabhu [1].

The transformation introduces the special form `future` and its companion procedure `touch` wherever it is deemed possible according to the analysis, making it brute-force in nature. The transformed program requires a runtime that supports `future/touch` constructs as follows:

- `(future e)` allows expression `e` to be evaluated immediately or at some unspecified point in the future —and possibly in parallel with the rest of the program. Either the value of `e` is returned, or a *promise* that represents the future value of `e`.
- `(touch f)` waits for and returns the value of the promise `f` or simply returns `f` itself if it is not a promise

The intentional semantics [2] of `future/touch` relieves the transformation from assessing whether it is beneficial to introduce them at each discovered opportunity. Instead, this responsibility can be left to the runtime. To illustrate the use of `future/touch`, the parallel `let` expression

```
(let|| ((a e1)
        (b e2))
  e_body)
```

can be implemented as follows:

```
(let ((f_a (future e1))
      (b e2))
  (let ((a (touch f_a)))
    e_body))
```

Using a concurrent runtime, we evaluate our transformation on well-known benchmark programs with respect to the following criteria:

1) **Completeness** how many opportunities for parallelization were exposed and subsequently exploited?
2) **Performance** does the parallelized program execute faster than the original sequential one?
3) **Scalability** does the performance of the parallelized program scale with the hardware support for parallelism?

The experiments demonstrate that our brute-force approach to parallelization realizes scalable speedups in certain benchmarks, while others would benefit from a more selective parallelization.

Although we descibe our technique in the setting of higher-order Scheme programs, it should be possible to transpose it to object-oriented programming languages. For

example, [3] discusses the link between the worlds of functional and object-oriented static analysis.

Section II relates parallelization opportunities to the different kinds of dependencies that can exist between two expressions. We describe how our transformation discovers and subsequently exploits opportunities for parallelization in Section III. Section IV discusses the concurrent runtime that is required by the transformed programs. Using this runtime, Section V evaluates our transformation on several well-known benchmark programs. We conclude this paper with related and future work in Sections VI and VII.

II. Dependencies and Parallelization

Consider parallelizing program P_1 depicted on the left-hand side of Figure 1. Potential targets for introducing parallelism are any point in P_1 at which a sequence of expressions is evaluated. For instance, the arguments `(h a)` and `(h b)` of the procedure application `(append (h a) (h b))` could be evaluated in parallel. Whether it is actually *safe* to evaluate these expressions in parallel, hinges on the *dependencies* among them. Expression e_2 is dependent on expression e_1 if e_1 must be evaluated before e_2 to guarantee the intended semantics of the program in which these expressions occur. Dependent expressions cannot safely be evaluated in parallel. Conversely, evaluating independent expressions in parallel cannot change the meaning of a program and hence it is safe to do so.

In program P_1, `(h a)` and `(h b)` can be evaluated in parallel because these expressions are independent. To this end, the aforementioned `future/touch` special forms can be introduced in procedure `appender`:

```
(define (appender h a b)
  (let ((ha (future (h a)))
        (hb (h b)))
    (append (touch ha) hb)))
```

In the transformed procedure, expression `(future (h a))` informs the evaluator that `(h a)` can be evaluated in parallel with the rest of the program. The `(future (h a))` expression evaluates to a so-called promise that represents the future value of `(h a)`. Expression `(touch ha)` causes the evaluator to wait for and return this value. This synchronizes the expression that is evaluated in parallel and the rest of the program. While expression `(h a)` is computed, the program can compute `(h b)`, so nothing would be gained by turning `(h b)` into a future computation as well.

Now consider program P_2 depicted on the right-hand side of Figure 1. In P_2, expressions `(h a)` and `(h b)` can no longer be evaluated in parallel. The latter expression depends on the former through the variable `z`. This variable is first read and updated during the evaluation of `(h a)` and subsequently during the evaluation of `(h b)`. Evaluating these expressions in parallel would therefore change the semantics of the program. Note that procedure `appender` in which expressions `(h a)` and `(h b)` reside, is identical in P_1 and P_2. Analyses that determine the dependences among expressions therefore have to consider the program as a whole.

In this paper, we describe a program transformation that introduces `future/touch` constructs to exploit all opportunities for parallelization that are deemed safe by such a whole-program dependence analysis. The transformation will parallelize the two applications of `h` in P_1, while it will correctly leave the corresponding applications in P_2 alone.

A. Types of dependencies

Dependencies can be categorized into *control* and *data* dependencies.

1) Control dependencies: Expression e_2 is *control-dependent* on expression e_1 if the result of evaluating e_1 determines whether e_2 will be evaluated or not. For example, in the expression

```
(if (not (zero? x))
        (f (/ y z)))
```

procedure `f` will only be called if the condition `(zero? x)` does not hold. Hence there is a control dependency from `(f (/ y x))` on `(not (zero? x))`. Control-dependent expressions cannot be reordered without changing the semantics of a program. We won't consider control dependencies any further in the remainder of this paper.

2) Data dependencies: Expression e_2 is *data-dependent* on expression e_1 if e_1 is evaluated before e_2 and e_2 accesses or modifies a resource that was accessed or modified before by e_1. In the context of this paper, a resource is a variable pointing to a fixed address that contains a modifiable value.[1] We distinguish between four types of data dependencies among consecutively executed expressions e_1/e_2, and give an example of each.

- *read/read dependency*: e_1 and e_2 share a variable they read. In P_1 and P_2 `(h b)` is read/read dependent on `(h a)` through `h`.
- *read/write dependency*: e_2 writes a variable that e_1 reads. In P_2 `(set! z (cons x z))` is read/write dependent on `(cons x z)` through `z`.
- *write/read dependency*: e_2 reads a variable that e_1 writes. In P_2 `(h a)` is write/read dependent on `(set! z (cons x z))` through `z`.
- *write/write dependency*: e_1 and e_2 share a variable they write to. In P_2 `(h b)` is write/write dependent on `(h a)` through `z`.

Read/read dependencies are harmless in the sense that they do not preclude evaluating the involved expressions in a different order. On the contrary, a data dependency that involves updating a variable prohibits evaluating the concerned expressions in parallel.

[1] We assume that there is a one-to-one mapping from variable identifiers to addresses. This can be ensured by applying α-conversion [4] to the input program.

```
1  (define (appender h a b)
2    (append (h a) (h b)))

4  (define (lister g)
5    (lambda (x)
6      (list (g x))))

8  (define (square x)
9    (* x x))

11 (appender (lister square) 42 43)
```

```
1  (define (appender h a b)
2    (append (h a) (h b)))

4  (define (lister g)
5    (lambda (x)
6      (list (g x))))

8  (define z '())

10 (define (conser x)
11   (set! z (cons x z)) z)

13 (appender (lister conser) 42 43)
```

Figure 1. Scheme programs P_1 (left-hand side) and P_2 (right-hand side) illustrating different types of data dependencies.

B. Determining data dependencies

Our program transformation has to be instantiated with an analysis that is capable of analyzing three sources of data dependence.

1) We need to determine the *data dependencies between expressions and variable binding operations* caused by special forms `define`, `let` and their variants. This kind of dependence is relatively straightforward to discover from the source code. For example, the body of `conser` on line 11 of P_2 cannot be evaluated before or in parallel with the `define` on line 8. The former depends on the latter since the body expects a binding for variable `z` to be present.
2) We need to determine the *intra-procedural data dependencies between expressions*. These data dependencies are also lexically evident from the source code and are therefore straightforward to discover as well. For example, it is immediately apparent that procedure application `(h a)` reads from variables `h` and `a`.
3) Lastly, we need an analysis that determines the *inter-procedural data dependencies between expressions*. Inter-procedural dependencies between expressions are, in contrast with the previous ones, challenging to determine. We already illustrated this for the *write/write* dependency of application `(h b)` on application `(h a)` in P_2. This dependency is not lexically apparent at either application site. There is no static control flow readily available for a higher-order program [5], and the variables written to or read from by invocation `(h a)` in general depend on the values of both `h` and `a`. The advanced dependence analysis by Might and Prabhu [1] is a suitable candidate for computing inter-procedural data dependencies between expressions in higher-order Scheme programs.

Our program transformation requires the dependence analysis to conservatively *over-approximate* the actual dependencies that will arise at run-time. Every dependency that may arise at run-time has to be included in the results, but the result may contain false positives. Some opportunities for parallelization might therefore remain unexploited because of a dependence that is a false positive, but – more importantly – all exploited ones are safe. Of course we require the analysis to support Scheme's higher-order procedures and side effects, so it can be applied to real-life Scheme programs. We will revisit the analysis in the implementation section of this paper.

C. Administrative Normal Form

In order to look for parallelization opportunities, we have to examine dependencies between expressions occurring in various syntactic constructs in an arbitrary Scheme program. To facilitate the detection of these opportunities, we can normalize source code into *administrative normal form* [6] (ANF). Complex expression are broken down into a sequence of simpler intermediate subexpressions by introducing unary (i.e., containing one binding) `let` expressions. For example, the expression for calculating the n-th Fibonacci number

```
(if (< n 2) n (+ (fib (- n 1)) (fib (- n 2))))
```

is normalized into the following expression:

```
(let ((_p0 (< n 2)))
  (if _p0
      n
      (let ((_p1 (- n 1)))
        (let ((_p2 (fib _p1)))
          (let ((_p3 (- n 2)))
            (let ((_p4 (fib _p3)))
              (+ _p2 _p4)))))))
```

In this example variables with prefix `_p` are unique variables introduced by the conversion process. In a similar manner ANF conversion also breaks down other types of conditional expressions, declarations, mutations and procedure bodies by explicitly naming subexpressions. As a consequence, instead of specifically treating all the separate syntactic cases for determining dependence, we can concentrate on one single construct after ANF conversion, namely a series of nested unary `let` expressions.

Because ANF conversion affects all types of Scheme expressions and terminates only when there are no more expressions to simplify, we argue that focussing on the

$$
\begin{array}{rcl}
u \in \mathsf{Var} & = & \text{set of identifiers} \\
q \in \mathsf{Quo} & = & \text{set of literals} \\
p \in \mathsf{Prim} & = & \text{set of primitives} \\
lam \in \mathsf{Lam} & ::= & (\lambda(u_1 \dots u_n)\ e_{body}) \\
proc \in \mathsf{Proc} & = & \mathsf{Lam} + \mathsf{Prim} \\
f, x \in \mathsf{Arg} & = & \mathsf{Var} + \mathsf{Quo} + \mathsf{Proc} \\
e \in \mathsf{Exp} & ::= & x \\
& | & (f\ x_1 \dots x_n) \\
& | & (\texttt{let}\ ((u\ x))\ e_{body}) \\
& | & (\texttt{let}\ ((u\ (f\ x_1 \dots x_n)))\ e_{body}) \\
& | & (\texttt{letrec}\ ((u\ lam))\ e_{body}) \\
& | & (\texttt{set!}\ u\ x) \\
& | & (\texttt{if}\ x_{cond}\ e_{cons}\ e_{alt})
\end{array}
$$

Figure 2. Administrative normal form (ANF) with recursive functions, mutable variables and conditional.

resulting series of nested unary `let` expressions is a sufficiently general approach to uncover data dependencies. For soundness, it is necessary to treat data dependencies at the level of *individual* expressions. However, this granularity may seem too fine-grained when it comes to parallelization. In the next section we will actually parallelize source code at a coarser scale.

Figure 2 depicts the abstract grammar of the ANF variant we will use throughout the remainder of this paper. Note how the operands at call sites are always "simple" expressions (i.e., a reference, literal, or lambda) and that procedure calls are either `let`-bound or in tail position.

III. Approach

Our approach for the parallelization of Scheme programs consists of transforming a program into its ANF equivalent, examine each series of unary nested `let` expressions for parallelization opportunities, and rewrite those nested `let` expressions by exploiting parallelism where possible. The granularity at which we examine parallelism is not a `let` binding expression, but rather a variable binding operation.

A. Dependence between variable binding operations

It is straightforward to reason about dependence on the level of variable binding operations: variable binding operation B_2 with binding expression e_2 is dependent on variable binding operation B_1 with binding expression e_1, if e_2 is dependent on B_1 or e_2 is dependent on e_1. By reasoning about binding operations and not individual `let` binding expressions, we can make our parallelism more coarse-grained. Instead of evaluating single binding expressions in parallel, we would like to group as many binding operations as possible to be evaluated in parallel. We will have more to say on grouping variable binding operations later.

Just as with expressions before, variable binding operations can be safely parallelized if and only if there are no *write/read*, *read/write* or *write/write* data dependencies between them. These dependencies are easy to detect using a dependence analysis with the proposed capabilities, although the following three points deserve additional attention.

1) Closures: Consider the following expression in which a closure is bound to a variable.

```
(let ((z #f))
  (let ((writez (lambda () (set! z 123) z)))
 ...))
```

Because the body of a lambda is not evaluated when the lambda expression is evaluated into a closure, we do not have to worry about any data dependencies that may arise on other binding expressions. However, we do have to take into account that the body of the lambda in our example has a dependency on the binding operation of variable `z`. This means that the binding operations of variables `writez` and `z` cannot occur in parallel.

2) Procedure invocations: The expression below binds the result of a procedure invocation to a variable.

```
(lambda (z)
  (let ((writez (lambda () (set! z 1) z)))
    (let ((squarez (lambda () (* z z))))
      (let ((newz (writez)))
        (let (zquared (squarez)))
          ...))
```

In this example, it is permitted to reorder the binding operations of variables `writez` and `squarez` because they are not dependent. But we cannot reorder the binding operations of variables `newz` and `squarez` because the body expressions associated with the two lambda expressions have an interprocedural *write/read* data dependency on each other through variable `z`.

3) Body reference: Besides data dependencies between variable binding operations, there is one more type of ordering dependency that we have to take into account. If we only want to parallelize expressions inside a series of nested `let`s without any effect on outside computation, we have to make sure that all future computations are terminated when the evaluator has finished evaluating the body expression and returns its value. In the expression

```
(let ((z 42))
  (let ((x (f z)))
    (* z z)))
```

we want the call `(f z)` to be finished before the value of body expression `(* z z)` is returned as the value for the entire expression. For this reason, and also to increase our chances of parallelization, we will always bind the body expression to a unique variable which we will call `body`, as shown below.

```
(let ((z 42))
  (let ((x (f z)))
    (let ((body (* z z)))  ; body variable
      body)))              ; =body reference
```

To distinguish between the variable `body` and a reference to it, we will denote the latter as `=body`. Even though `(f z)` does not contribute to the end value of the above expression,

and even though it can be reordered with `(* z z)`, we still require that `=body` is dependent on the binding operation of variable `x`. In general, we make the body reference expression `=body` dependent on any binding operation on which no other binding operation is dependent. This constraint ensures that the body reference expression `=body` is always the unique last expression in all dependency chains that we can construct for a series of nested `let` expressions. In practice, binding expressions in `let` expressions that are evaluated for side-effects only and that do not (indirectly) contribute to the return value of the `let`, are rare.

B. Dependency graphs

We can use dependency information to construct a *binding dependency graph* that models the temporal evaluation constraints imposed by the dependencies on variable binding in series of unary `let` expressions. This binding dependency graph is a directed, acyclic graph (DAG) in which each node represents one variable binding operation from the nested lets. It also includes the *body node* that represents the body reference `=body`. The body node is always the unique sink node in a binding dependency graph.

Figure 3 shows an example of a dependency binding graph for a series of nested unary `let` expressions. Most dependencies in the graph are dependencies between binding expressions and variable binding operations. The dependency of `p4` upon `p3` is the most interesting dependency because it expresses an interprocedural dependency resulting from the two procedure invocations `(writez)` and `(readz)`.

For more complicated series of unary `let`s, binding dependency graphs are not always the most straightforward representation for deciding the optimal order of variable binding with respect to parallelization. To make this binding order explicit, we can apply simple graph transformations like pruning and grouping.

- *Pruning* is the operation of removing transitive edges. If we are interested in the order in which variables are bound, transitive edges are superfluous. In Figure 3b, the dependency of `body` on `p3` can be pruned because the dependency chain of variables `p3` → `p4` → `body` ensures that variable `body` is always bound after variable `p3`. Pruning increases the opportunities for grouping.
- *Grouping* is accomplished by taking connected paths of nodes with exactly one entry point and one exit point. Because we group variable binding operations that must always be evaluated in a particular sequential order, grouping is equivalent to basic block creation in traditional control flow diagrams. The fact that grouping makes the introduced parallelism more coarse-grained is not merely an additional benefit, but essential if we want to avoid unnecessary synchronization points during parallel evaluation, which could result in excessive overhead. Figure 4b shows the effect of grouping.

The result after pruning and grouping is a binding *order* graph, which is a more akin to a task dependency graph in which a task consists of one or more variable binding operations. The unique body node is like other nodes but additionally contains the body reference `=body` as its last element.

C. Generating code

Once we have exhaustively applied pruning and grouping, the resulting binding order graph is in a form that allows straightforward conversion into Scheme code. To determine the final variable binding order, we divide the graph into layers by applying a *topological sort*. In a topological sort every node is assigned a rank or level. The level of a node is equal to the number of nodes in the *longest* path possible from that node to the (unique) sink node. The sink node trivially has level 1.

The key insights for generating code from a binding order graph are the following:

1) All variable binding operations inside a single node must be evaluated sequentially (because of the way we defined the grouping operation).
2) A group of variable binding operations contained inside a single node can be evaluated in parallel with other groups of binding operations in nodes of the same level.

The intuition behind the second insight is that every *branch* in the binding order graph can be translated by using a `future/touch` construct in the parallelized version. Figure 4c illustrates the code that can be generated from a binding order graph.

IV. Implementation

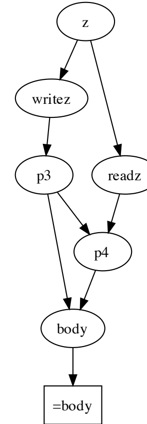
We have implemented all the ideas and techniques presented in this paper. The resulting software artefact is a Scheme evaluator that is capable of performing automatic parallelization. The evaluator is instantiated with a static dependence analysis and a runtime capable of handling `future/touch` constructs. It takes sequential Scheme programs as input and transforms them into their parallel versions. When executed on parallel hardware, the runtime effectively takes advantage of multiple processing cores when evaluating expressions concurrently. The evaluator, including the dependency analysis and runtime, are implemented in Java. It is publicly available at `http://code.google.com/p/streme/source/browse/\#svn/tags/scam11`.

A. Concurrency constructs

The concurrent runtime understands a subset of R5RS Scheme, together with some additional constructs for concurrency. The primitive concurrency constructs are `touch` and `future`, which behave like true annotations having intentional semantics. This means that `future` is not simply

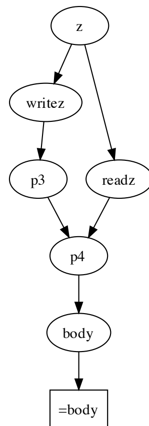
```
(let ((z 0))
  (let ((writez (lambda () (set! z 123))))
    (let ((readz (lambda () z)))
      (let ((p3 (writez)))
        (let ((p4 (readz)))
          (cons p3 p4))))))
```

(a) A series of unary `let` expression.

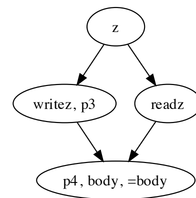


(b) Corresponding graph.

Figure 3. Dependency binding graph for a series of nested unary `let` expressions. Binding nodes are oval-shaped. The body reference node is depicted in a rectangular shape. An edge between two nodes means that the target node is dependent on the source node.



(a) After pruning.



(b) After grouping.

```
(let ((z 0))
 (let ((writez undefined))
  (let ((fp3 (future
              (begin
               (set! writez (lambda ()
                             (set! z 123)))
               (writez)))))
   (let ((readz (lambda () z)))
    (let ((p3 (touch fp3)))
     (let ((p4 (readz)))
      (let ((body (cons p3 p4)))
       body)))))))
```

(c) Corresponding code.

Figure 4. (a). The binding dependency graph of Figure 3 after pruning dependency `p3` $\rightarrow$ `body`. (b) Binding order graph obtained from (a) by grouping. (c) The code that is generated for the graph in (b).

`fork` or `spawn` for example, in which case evaluation is unconditionally started in another thread and `touch` plays the role of a `join` that returns a value. `future/touch` is not `delay/force` either, computing the value of an expression at the point where it is needed. And both `future` and `touch` are not simply the identity function `id`, the function that immediately returns its evaluated operand. In theory `future` and `touch` can have any of the previous three semantics, and this at the discretion of the runtime. In practice, our runtime will only apply `fork` or `id` semantics:

- It will strive for maximizing parallelism by using `fork` semantics whenever the number of actively running future computations does not exceed a certain preset threshold. In this case `future` returns a promise.
- It will fall back to `id` semantics whenever it decides not to initiate a future computation. `future` then returns the value of its operand.

The intentional semantics of `future` and `touch` are not chosen by accident, but are actually indispensable for our brute-force approach to work. Every initiation of a future

computation using `future`, managing future computations, and sychronizing their results through `touch`, incurs a runtime cost. This overhead may outweigh the possible performance gains realized by parallelization, especially in highly recursive code. The intentional semantics allow us to *not* care about the possible overhead parallelism may incur, by leaving it up to the runtime system to decide to actually set up a future computation or not.

B. Architecture

The architecture of our evaluator follows a typical pipeline design. The pipeline consists of various analyzers, transformers and compilers that are chained together and each do a piece of the work. The pipeline can be divided into a front-end, middle-end and back-end.

The front-end is responsible for parsing and macro expansion, and produces an abstract syntax tree (AST).

The middle-end contains infrastructure to analyze and transform ASTs. First, the AST is converted into ANF. Then static analysis on all series of nested unary `let`s is conducted. The result of this analysis is one binding order graph per structure of nested `let`s. These graphs form the basis for deciding if and how to rewrite the `let`s to introduce parallelism.

The back-end is a concurrent Scheme runtime that understands ANF and `future/touch`. It uses the Java fork/join-framework [7] that employs a work-stealing strategy to execute submitted tasks. This framework is scheduled for inclusion in Java 7, but already available as a separate JAR file for Java 6. When a future computation is submitted as a task to a `java.util.concurrent.ForkJoinPool`, the returned instance of `Future` serves as a promise value. The procedure `touch` checks if it receives a `Future` as argument. If so, it performs a blocking `get()` on its argument, else `touch` simply returns its argument.

V. Experiments and discussion

In order to validate our brute-force automatic parallelization approach, including the underlying static analysis and concurrent runtime, we carried out experiments by parallelizing and running several well-known Scheme benchmarks. The experiments were carried out on a Sun Java 1.6.0 runtime environment with a HotSpot 64-bit server JVM running on an Intel machine with 8 Xeon processors. Because of hyperthreading, the number of reported processors by a call to `Runtime.getAvailableProcessors()` is actually 16.

The underlying dependence analysis is based on k-CFA. For running the benchmarks we set its sensitivity $k = 4$. Lower values for k resulted in some code paths not being taken. Higher values did not have a noticeable impact on the results.

We briefly describe some of the benchmarks we used in our experiments and will discuss here, in the table below.

`fib`	Tree-recursive version of the function that calculates the n-th Fibonacci number, with `(< n 2)` as condition for termination. For our benchmark $n = 40$.
`tak`	Function involving triple recursion in the version of McCarthy, returning `z` when `(not (< y x))`. The three arguments used are 40, 30 and 20.
`quicksort`	Straightforward implementation of Hoare's sorting algorithm, applied to a fixed list of 500,000 generated pseudo-random numbers.
`nboyer`	Benchmark that is more representative of real programs performing symbolic computation, employing rule-directed rewriting. We use $n = 3$.
`nqueens`	Combinatorial problem requiring the placement of n queens on a $n \times n$ chessboard so that no two queens attack each other. We take $n = 12$.

A. Safety and completeness

We have not constructed a formal proof of the correctness of our approach. Assuming that the underlying dependence analysis from [1] is sound is already a big step in the right direction, considering that this analysis handles the hardest case of dependence as a consequence of procedure invocation. Because the remaining lexical dependence analysis is less complex and our dependency graph transformations are straightforward, we are confident that a more formal proof of the safety and completeness of our approach is feasible.

Instead of a formal proof, we assessed safety and completeness by creating and maintaining a batch of unit tests that test the safe parallelization of all opportunities present in small test programs that cover the different cases. Additionally, for small programs and benchmarks manual verification is also possible. For larger benchmarks like `nboyer` manual parallelization, and the verification thereof, becomes tedious and error-prone. However, because our approach scales from small test programs to large ones (the unit of testing is a single series of nested unary `let`s), together with the fact that we test the correctness of the values produced by parallelized benchmarks when run on parallel hardware, suggests that our approach is safe and covers all parallelization opportunities.

The number of interprocedural dependencies inside nested `let`s per benchmark, detected by static analysis, is rather low: 0 for `fib`, 2 for `tak`, 6 for `quicksort` and `nboyer` and 12 for `nqueens`. Of course, proving the *absence* of this type of dependencies remains crucial in our approach.

B. Scalability

For measuring scalability we examine the impact of the number of available processing cores on the running time of our parallelized benchmarks. Our experiments are carried out on a JVM. As a result, there is no way to know whether each thread is mapped to a different CPU when available [7]. Furthermore, the `ForkJoinPool` instance used by our concurrent runtime is initialized with a parameter called `parallelism` that indicates the *target* number of worker threads used by the pool. Internal documentation of the class states that "it is impossible to keep exactly the target (parallelism) number of threads running at any given time" and that "heuristic guidance" is required. However, monitoring of CPU activity while executing the benchmarks, together

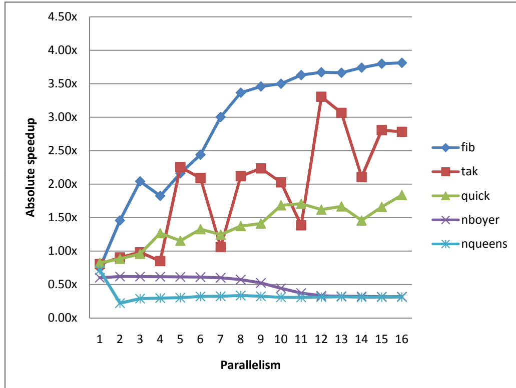


Figure 5. Absolute speedup of parallelized benchmarks as a function of parallelism.

with the data from micro-benchmarks like `fib`, show that the number of processors used by the JVM increases as the parallelism increases.

For running our benchmarks we established the threshold for creating future computations at $2(n-1)$ for parallelism n. With this value we keep the number of future computations under control so the runtime does not get swamped with too many tasks, creating overhead. On the other it ensures that there are enough tasks so that the worker threads can keep themselves busy.

Figure 5 shows a graph of the speedup of our benchmarks plotted against the level of parallelism. In this figure we see that micro-benchmark `fib` scales well when parallelism increases. Benchmarks `tak` and `quicksort` also scale, but not as good as `fib`. These three benchmarks are implemented with divide-and-conquer algorithms and clearly benefit from our transformation. The reason is that these algorithms result in a computation tree at run-time consisting of many independent tasks. Furthermore, the longest running tasks are created in the beginning of the computation, when the limit on the number of concurrently executing future computations has not yet been reached.

`nboyer` and `nqueens` don't scale at all when parallelized, but show a significant performance degradation that stabilizes at a certain degree of parallelism. Closer examination of the parallelized `nboyer` and `nqueens` programs reveals that the expressions wrapped inside `future`s are very simple. Typical examples include taking the `car` or `cdr` of a list, or subtracting two numbers. Clearly the overhead of evaluating many short-running expressions in parallel has a negative impact on the running time. As expected, this overhead increases as the degree of parallelism increases.

C. Performance

The goal of parallelization ultimately is to get a speedup in performance when parallel programs are executed on parallel hardware. Let T_s be the time it takes to execute the unparallelized ANF-converted benchmark in serial, and $T_p(n)$ the time it takes to execute the parallelized benchmark with parallelism n. The absolute speedup for parallelism n then is given by the quotient $T_s/T_P(n)$.

The execution time $T_p(1)$ of a parallelized benchmark is larger than the running time of its sequential version T_s, although we perform some limited post-processing on the parallelized code. This can be seen in Figure 5, where for all benchmarks the absolute speedup is below 1 for $n = 1$. For parallelized benchmarks that do not scale, it automatically follows that for parallelism $n > 1$, $T_p(n)$ will never be smaller than T_s. For parallelized benchmarks that do scale well, $T_p(n)$ will eventually dip below T_s as the degree of parallelism increases.

The reason for the slowdown in execution time $T_p(1)$ versus T_s is that the subsequent program transformations that transform a sequential Scheme program into a parallel one, introduce additional syntax. This negatively impacts execution time, although we believe that there is still room for improvement by applying more aggressive post-processing on the parallelized code than is now the case.

D. Heuristic

A recursive divide-and-conquer program should scale well when parallelized with our approach. However, the `quicksort` benchmark did not scale as well as the `fib` benchmark when parallelism increased, although both are implemented as a tree-recursive algorithm.

A second observation was that the process of parallelizing a sequential program introduced extra syntax that has a negative impact on the running time of a program.

To see if we could lower the running time and improve the scalability of our parallelized `quicksort`, we digressed a little from our purely brute-force approach by experimenting with a very simple heuristic.

The heuristic is applied on a binding order graph (Section III-B) and works as follows:

- Fold all nodes that don't apply non-primitive procedures into one single node.
- All nodes that do contain non-primitive procedure invocations are left untouched.

The effect of applying this heuristic is that we only introduce a `future` construct when a non-primitive procedure invocation is involved. We only work with non-primitives (i.e., user-defined procedures) because the invocation of a primitive usually does not result in a deep computation tree at run-time. Alternatively, for primitives it is possible to do a more specific case-by-case analysis to see which invocations are worth parallelizing. Note that for this heuristic we need the value flow from the static analysis in order to determine the type of procedures that flow to operator positions.

Figure 6 shows the impact of this heuristic on the performance of the parallelized `quicksort` benchmark. Applying the heuristic is not only beneficial in terms of scalability,

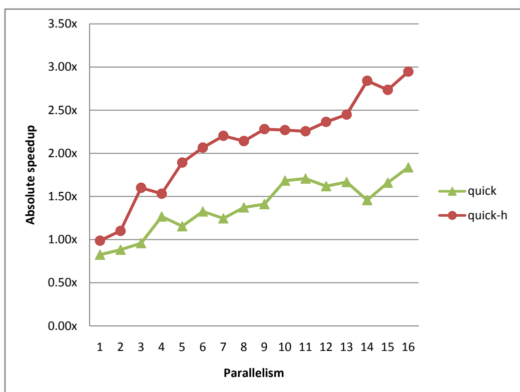


Figure 6. Absolute speedup of `quicksort` benchmark, with (`quick-h`) and without (`quick`) heuristic applied.

it also reduces the running time. For parallelism $n = 1$ the running time decreases from 65.9 seconds to 55.1 seconds, or a 84% improvement. For $n = 16$, the running time decreases from 29.6 seconds to 18.5 seconds, which is still nearly a 63% improvement.

Applying the heuristic on the `nqueens` benchmark also had a significant impact. The heuristic effectively rejected all the parallelization opportunities present in the code. As a consequence, the "parallelized" `nqueens` benchmark was essentially unchanged from the original sequential benchmark. Therefore it did not suffer from the performance degradation as did `nqueens` when parallelized without the heuristic applied (depicted in Figure 5). Of course, no speedup was realized either.

Applying the heuristic on other benchmarks had no impact. `fib` and `tak` always feature a non-primitive procedure invocation, and `nboyer` had a shorter running time for $n < 8$ but exhibited the same performance degradation afterwards.

VI. Related work

Our research builds on the work by Might and Prabhu [1]. They present an interprocedural static analysis on which we rely to decide whether it is safe to parallelize certain expressions in a program. In [1] the authors also briefly touch upon the question of whether the introduction of parallelization at some point is beneficial or not, although it is never explored any further. We do not directly answer this question in our brute-force approach ourselves, but clearly make a choice by basically ignoring this factor, with the exception of experimenting with a simple heuristic to only parallelize non-primitive procedure applications.

Might and Prabhu were inspired by the work of Harrison, who designed the Parcel and Miprac compilers that perform interprocedural dependence analysis and program restructuring to automatically parallelize Scheme programs [8], [9]. An important difference with our approach is that Parcel relies on an unstructured and complex intermediate form that bears little resemblance to the original input program. As a consequence, Parcel's parallelization algorithms are also very complex and difficult to extend or adapt. In contrast, our approach is very straightforward because it uses ANF as an intermediary form. ANF is a subset of Scheme and as such it is already well-understood and probably easier to analyze and manipulate in a Scheme context.

Halstead's Multi-Lisp extended Scheme for expressing concurrency and introduced the concept of *futures* into Lisp [10]. `pcall` is a construct in Multi-Lisp that concurrently evaluates the arguments of a procedure invocation in a fork/join style. Halstead recognizes that `pcall` can be rewritten using futures, similarly to how we proposed to implement `let||` as a standard `let` using futures. He also remarks that the "use of `pcall` within a recursive procedure such as a tree walk can result in exponential amounts of parallelism", which agrees with the excellent results we observe for highly recursive divide-and-conquer benchmarks in our experiments.

One alternative approach to parallelization is manual parallelization. However, it is generally accepted that in general manual refactoring is tedious and error-prone. This is worsened by the fact that the predominant model of concurrent programming today is based on multithreading, which is a very difficult to comprehend by programmers due to the intrinsic non-determinism that threads introduce [11].

In automatic dynamic parallelization, the compiler or interpreter is tasked with introducing parallelism into a sequential program [12] at runtime. The most obvious difference with our approach is that the necessary analysis also takes place at runtime. Static and dynamic analysis can work together to increase their usefulness.

VII. Future work

The experience gained by designing and implementing our brute-force parallelization technique has inspired us to start building an improved version based on the same ideas but with more focus on usability and practicality. The idea is to investigate the implementation of an analysis that does not require an a priori code transformation like ANF conversion. The results from this analysis then can be used in source-to-source program transformations that require not only the preservation of program semantics but also program syntax. Practical applications that spring into mind are IDEs that are able to flag concurrent constructs that are unsafe, or are able to indicate opportunities for parallelization that are unexploited.

ANF conversion and the parallelization of nested `lets` introduces additional syntax which hampers the absolute speedup of the parallelized program. In order to address this situation, we intend to apply additional post-processing on the resulting parallelized code to reduce its runtime.

Our implementation of the underlying static analysis has some limitations. First of all, we do not model the `car` and

`cdr` separately but consider a cons cell to be one resource, which incurs a precision penalty when specifically reading from or writing to the `car` or `cdr`. Similarly, we don't model the individual elements of a vector. In future work, we want to increase the precision of the analysis by being able to reason over these individual elements of cons cells and vectors. At the moment, reading from a cons cell or vector always returns a placeholder value ("any"). Another limitation in the analysis is the fact that it currently cannot handle this "any" value when it occurs as a possible value of an operator.

VIII. Conclusion

We described a brute-force technique for the parallelization of Scheme programs containing higher-order procedures that perform side effects. Our approach starts by transforming the input program into administrative normal form. This serves two purposes: it introduces many series of nested unary `let` expressions that are the target of our automatic parallelization approach, and it renders the program in a form that is suitable for dependence analysis. The key idea is to look at every series of nested `let` expressions, and decide where it is possible to evaluate binding expressions in parallel using state-of-the-art static dependence analysis. To generate code we construct dependency graphs that model the dependencies inside a series of nested unary `let` expressions. We transform these graphs in such a way that it becomes evident at which points `future` and `touch` can be introduced. We also described the intentional semantics that `future/touch` must possess in a concurrent runtime on which the parallelized programs will be executed.

We implemented an evaluator that employs our brute-force parallelization technique, together with a concurrent runtime that provides `future/touch` with the required intentional semantics, and applied it to a number of benchmarks.

The conclusion from our experiments is that it is feasible to use a brute-force approach, backed by a static analysis, to safely and automatically parallelize Scheme programs. The parallelization is effective for programs that contain recursive divide-and-conquer algorithms, which scale well when executed on parallel hardware. Our experiments suggest that our approach can benefit from complementary techniques for parallelizing more general Scheme programs.

Acknowledgments

The authors thank Matthew Might for his useful comments and suggestions during the implementation of the analysis. Jens Nicolay is funded by the *COGNAC* project sponsored by the "The Research Foundation - Flanders" (FWO Vlaanderen). Coen De Roover is funded by the *Stadium* SBO project sponsored by the "Flemish agency for Innovation by Science and Technology" (IWT Vlaanderen). This research is partially supported by the IAP Programme of the Belgian State.

References

[1] M. Might and T. Prabhu, "Interprocedural Dependence Analysis of Higher-Order Programs via Stack Reachability," *Technical Report CPSLO-CSC-09-03*, p. 75, 2009.

[2] C. Flanagan and M. Felleisen, "The semantics of future and its use in program optimization," in *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 1995, p. 220.

[3] M. Might, Y. Smaragdakis, and D. V. Horn, "Resolving and exploiting the k-cfa paradox: Illuminating functional vs. object-oriented program analysis." in *Proceedings of the 31st Conference on Programming Language Design and Implementation (PLDI 2006)*, Toronto, Canada, June 2010, pp. 305–315.

[4] G. E. Revesz, *Lambda-calculus, Combinators and Functional Programming*, 1st ed. New York, NY, USA: Cambridge University Press, 2009.

[5] O. Shivers, "Control flow analysis in scheme," *ACM SIGPLAN Notices*, vol. 23, no. 7, p. 174, 1988.

[6] C. Flanagan, A. Sabry, B. Duba, and M. Felleisen, "The essence of compiling with continuations," *ACM SIGPLAN Notices*, vol. 28, no. 6, pp. 237–247, 1993.

[7] D. Lea, "A Java fork/join framework," in *Proceedings of the ACM 2000 conference on Java Grande*. ACM, 2000, pp. 36–43.

[8] W. Harrison III, "PARCEL: Project for the automatic restructuring and concurrent evaluation of lisp," in *Proceedings of the 2nd international conference on Supercomputing*. ACM, 1988, p. 538.

[9] W. Harrison III and Z. Ammarguellat, "PARCEL and MIPRAC: Parallelizers for Symbolic and Numeric Programs," 1992.

[10] R. Halstead Jr, "Multilisp: A language for concurrent symbolic computation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 7, no. 4, pp. 501–538, 1985.

[11] E. Lee, "The problem with threads," *Computer*, vol. 39, no. 5, pp. 33–42, 2006.

[12] C. Herzeel and P. 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*, ser. OOPSLA '10. New York, NY, USA: ACM, 2010, pp. 377–396. [Online]. Available: http://doi.acm.org/10.1145/1869459.1869491