Dynamic Parallelization of Recursive Code

Part I: Managing Control Flow Interactions with the Continuator

Charlotte Herzeel Pascal Costanza

Software Languages Lab, Vrije Universiteit Brussel, Belgium

charlotte.herzeel/pascal.costanza@vub.ac.be

Abstract

While most approaches to automatic parallelization focus on compilation approaches for parallelizing loop iterations, we advocate the need for new virtual machines that can parallelize the execution of recursive programs. In this paper, we show that recursive programs can be effectively parallelized when arguments to procedures are evaluated concurrently and branches of conditional statements are speculatively executed in parallel. We introduce the *continuator* concept, a runtime structure that tracks and manages the control dependences between such concurrently spawned tasks, ensuring adherence to the sequential semantics of the parallelized program. As a proof of concept, we discuss the details of a parallel interpreter for Scheme (implemented in Common Lisp) based on these ideas, and show the results from executing the Clinger benchmark suite for Scheme.

Categories and Subject Descriptors D.1.3 [*Software*]: Programming Techniques—Parallel Programming; D.3.4 [*Software*]: Programming Languages—Interpreters

General Terms Languages, Performance

Keywords Recursion, automatic parallelization, software speculation, virtual machines, continuator

1. Introduction

Multicore processors are sneaking into our everyday life: Desktops and laptops have been equipped with multicore processors for several years now. In general, if we want our programs to take advantage of such processors, we can choose to adopt parallel programming, which is, however, known to be very difficult and was until recently only considered to meet firm performance requirements. Most experience with parallel programming stems from highly specific applications in the domains of scientific computing and multimedia, which run on specialized hardware or clusters [18]. It is not clear whether these experiences can be easily transferred to a desktop setting with multicores.

It is in this light that a new family of multiprocessing languages is emerging, examples of which include Sun's Fortress, IBM's X10, and Cray's Chapel. These languages are in turn based on experiences with academic multiprocessing languages such as Cilk, Multilisp, Qlisp, and so on. In such languages, the programmer merely exposes the potential parallelism in the program. It is the language's *runtime system* that is then responsible for generating threads and scheduling them efficiently on the hardware. The results are promising, but it may still take years for languages like Fortress, X10, and Chapel to be fully developed, and even if we get them, there is the problem that we need to rewrite all existing applications from scratch if we want them to take advantage of multicores.

In contrast, with *automatic parallelization*, the compiler is fully responsible for mapping programs onto threads [37]. An important observation is that most work on automatic parallelization focuses on loops [4, 9, 10, 12, 13, 37, 52, 54], but this is insufficient when considering recursive code, because there is an important class of algorithms that cannot be easily and efficiently expressed as loops [1, 44]. Such recursive code is abundant in programs written in functional languages, where recursion is a favorite coding pattern. Consequently, there is an important class of programs that are not automatically parallelized using existing techniques.

Our position is that we should also come up with new virtual machines that can parallelize the execution of existing sequential languages. In this paper, we investigate one aspect of such a virtual machine, namely how to automatically extract and manage the parallelism in recursive code. More concretely, we investigate a virtual machine to automatically extract parallelism by combining parallel argument evaluation and speculative branching, which were introduced as language constructs in earlier languages [21, 28, 47]. We discovered that building parallel argument evaluation and speculative branching into a virtual machine requires a runtime mechanism for managing the control flow of the different

OOPSLA/SPLASH'10, October 17–21, 2010, Reno/Tahoe, Nevada, USA.

tasks that are spawned to execute a program. For this, we developed the *continuator* concept, a structure that tracks the dependences between the spawned tasks. The continuator manages the interactions between parallel argument evaluation, speculative branching, and side effects. The continuator also solves one of the long-standing problems in implementing speculative computation: It makes it possible to squash wrongly speculated computations without freezing all interpreter execution and guarantees that wrongly speculated computations do not keep spawning new tasks. As we explain in the rest of the paper, the idea is to represent the execution trace of a computation as a tree, where we use continuators to represent its nodes.

As a proof of concept implementation, we present a parallel interpreter for Scheme that implements our approach. We choose Scheme because it is a minimal language with all the basic ingredients for expressing different programming paradigms [1]: In Scheme one can write anywhere on the spectrum from very functional code (which is easier to parallelize) to very imperative code (which in general cannot be done in parallel). As a validation, we present the results of our interpreter running the Clinger benchmark suite for Scheme [14]. For functional code, the numbers show good speedups compared to a sequential implementation, despite the fact that our parallel interpreter uses a very straightforward task scheduler. This proves that the overhead of the continuator infrastructure can be sufficiently minimized and is in an acceptable range. In order to ensure adherence to the sequential semantics of the interpreted language, the continuator currently guarantees the absence of data races by synchronizing the computation at each assignment. As a consequence, code with many side effects currently runs in constant time with respect to the number of processors, but we present suggestions for improving this in future work.

The contributions of this paper are:

- a dynamic approach for automatically parallelizing recursive code, which combines parallel argument evaluation and speculative branching
- the *continuator* concept, which manages the control flow interactions between parallel argument evaluation, speculative branching, and side effects
- a squashing mechanism that guarantees that wrongly speculated computations are fully discarded, without freezing the program execution
- a parallel Scheme interpreter as a proof of concept implementation of our approach
- benchmarks showing that the overhead of our continuator infrastructure is in an acceptable range

2. Toward dynamic parallelization

Over the past years, several new parallel programming languages were developed that attempt to hide some of the complexity of parallel programming. These programming languages provide language constructs that enable the programmer to expose the parallelism in an application, but it is the responsibility of the language's runtime system to generate threads and schedule them efficiently on the hardware. We next discuss how parallelism is typically extracted from recursive programs in these languages, namely by calling the recursive subcalls in parallel, and by parallelizing the execution of conditionals.

2.1 Extracting parallelism from recursive code

As an example of a recursive program, consider Fig. 1, which shows the definition of a procedure for counting the leaves of a tree structure (in Scheme). It is a tree-recursive program, given the two recursive calls in the else branch. If we can execute the recursive subcalls in parallel, we are likely to gain a speedup.

```
(define (count-leaves x)
  (cond ((null? x) 0)
        ((not (pair? x)) 1)
        (else
         (+ (count-leaves (car x)) (count-leaves (cdr x))))))
```

Figure 1. Recursively counting the leaves of a tree

Multiprocessing languages like Multilisp, Qlisp, and Cilk use such examples to motivate the need for a *future*, *fork-join*, or *parallel call* construct, which makes it possible to call a procedure so that its arguments are processed in parallel instead of sequentially. In our example, the `+` should be called thusly to make sure that the two recursive calls to `count-leaves` execute in parallel.

Another approach for further increasing the available parallelism in a recursive program is *speculative computation* [45, 47, 50]. The idea behind speculative computation is to start executing computations before they are known to be required. For example, speculative computation can speed up execution of a search algorithm when all of the branches in the search space are explored in parallel, even though only one of them yields the correct result [46]. To support speculative computation, languages like Multilisp and Qlisp introduce parallel variants of constructs like `and`, `or`, and `if`. QLisp additionally suggests the concept of *heavyweight futures* as a means to express speculative computation [25].

As an example, consider Fig. 2. We adapted this example from Russell and Norvig's textbook on Artificial Intelligence (AI) (see p. 73 [55]). It illustrates the general pattern of AI search algorithms. The idea is that the search space is represented as a tree of nodes to be explored. The code implements a loop that traverses the nodes in the search space (here stored in a queue). During each iteration, the code checks whether a particular node satisfies the goal of the search, see `goal-test?` in the second if expression. If the node satisfies the test, the loop returns it as a result. Otherwise, new nodes are generated (`expand`) and the search loop continues. Goal testing and search space expansion are

typically very costly operations, and it is reasonable to assume that parallelizing their execution yields a performance improvement. In Multilisp, for example, this can be achieved by marking the `if` as speculative.

```
(define (general-search problem queuing-fn)
  (define (loop node nodes)
    (if (empty-queue? nodes)
        node
        (let ((fnode (remove-front nodes)))
          (if (goal-test? problem (node-stage fnode))
              fnode
              (loop fnode
                    (queing-fn nodes (expand fnode problem)))))))
  (loop '() (make-initial-queue problem queuing-fn)))
```

Figure 2. Recursively searching a problem space

2.2 Difficulties and open issues

The proposed constructs for parallelizing recursive programs are quite straightforward, but implementing *and* using them efficiently is not trivial.

One potential problem with language constructs for parallel procedure calling and speculative computation is that they lead to programs that produce a lot of fine-grained parallelism. It is in general more difficult to schedule fine-grained parallelism than coarse-grained parallelism efficiently on multicores because each parallel computation needs to be represented in memory, needs scheduling time, and so on [30]. Over the years, several runtime architectures and scheduling strategies were developed to properly handle this. The literature argues that a runtime architecture based on *work-stealing* scheduling provides the most efficient solution to the granularity problem. It is, for example, used in Multilisp [28], Qlisp [49], Cilk [8], Fortress [11], and X10 [3]. We discuss work stealing in more detail in Section 3.1.

A related problem is that the effectiveness of parallelizing parts of a program depends on that program's input. For example, it is interesting to parallelize the execution of `(fib 35)`, which computes the Fibonacci of 35, but it is less interesting to parallelize the execution of `(fib 2)`. The programmer must take this into account. The Cilk programming environment, for example, comes with a tool that monitors executions and reports programmers how much potential parallelism exists in a particular computation (see Chapter 15 of [35]). Similar experiments were done with Qlisp where the programmer estimates the threshold for switching to parallel execution [24], but finding the right threshold is a matter of trial and error.

Another issue is that—in all parallel programming languages we are aware of—interactions between side effects and parallel language constructs must be handled explicitly by the programmer. That is, side effects must be explicitly synchronized using locks or atomic blocks, and in case of errors, it is the programmer's responsibility to properly restore state. This is particularly difficult in combination with speculative computation: When a speculative computation performs side effects, and it turns out that the speculative computation is wrong, the side effects that were already performed must be undone by the programmer [45, 47].

Using speculation can also severely slow down applications when the program consistently speculates on the wrong computations. In Multilisp, programmers can influence which computations are favored for speculation by setting task priorities, but this considerably complicates the code [47]. Another issue is that wrongly speculated computations may never be deleted and as such swamp the processor with unnecessary work [45–47, 50]. See Section 7.4 for a more detailed discussion on this.

2.3 A virtual machine approach

We think that most of the above issues should be handled implicitly by the virtual machine. Our long-term goal is to build a virtual machine that automatically parallelizes sequential code, steered by advanced runtime analysis [31]. Such a virtual machine would perform hot-spot analysis to detect which parts of the program are interesting to parallelize and decide where to insert parallel procedure calls or speculative computations. It could furthermore monitor the inputs of programs and decide to switch to sequential execution for short computations. In case of parallel conditionals, a virtual machine could keep track of which branches are most often taken and perform branch prediction to avoid speculating on the wrong branches. The interactions between side effects and parallel constructs would also be handled automatically. For this, dependence analysis [37] could be included as part of the dynamic compilation phase to detect and remove potential data races. And so on.

This paper is the first step to such a parallelizing virtual machine and investigates how we can automatically extract and coordinate parallelism in recursive programs. As an experiment, we build an interpreter for Scheme that executes procedure calls and conditionals *by default* in parallel. Our contribution is the *continuator* concept, a mechanism that automatically manages the *control flow interactions* between parallel procedure calling, speculative computation, and side effects. It also solves the problems that are present in existing implementations of speculative computation: Our system automatically deletes wrongly speculated computations and makes sure that they cannot loop forever. Adding hot-spot analysis and dynamic compilation is part of future work and is discussed in Section 6.

3. Parallel execution by default

Our research approach has been to test our ideas in a straightforward language implementation in the form of an interpreter. This approach is in line with the Lisp tradition for using interpreters to understand execution semantics and using them as a starting point to develop new program analysis techniques that can be used in compiler and virtual machine implementations [19, 51, 62]. The most well-known exam-

ple where this approach is used are the *lambda papers* [22, 57–59, 61–64], a series of papers by Sussman and Steele that use interpreters to explore advanced programming concepts and implementation techniques, and led to the discovery of techniques for optimizing tail recursion [59], discussed solutions to the *funarg* problem [62], and showed that procedure calls do not necessarily impose an overhead compared to gotos [58]. Similarly, aspect weavers [39, 43], Haskell [34], Java [27], JavaScript [33], and Smalltalk [36]—to name just a few—were all first implemented as interpreters.

Our interpreter is an AST-based interpreter, rather than one based on byte codes. The advantages and disadvantages of ASTs compared to byte codes are subject to controversy [17, 40]. We believe that AST-based interpretation is more amenable to parallelization, because it gives rise to a tree-recursive process [1], which potentially yields exponential amounts of parallelism [28], whereas the flat representation of byte codes gives rise to an iterative, sequential interpretation process.

Our interpreter implements a non-trivial subset of Scheme, close to the R^{3}RS specification of Scheme[1], but does not support `call/cc` nor I/O operations. I/O operations are not particularly relevant for our experiments, and call/cc is not supported because it is not clear what a construct for reifying the program stack should return in a parallel interpreter, where the stack is spread over multiple threads. Nonetheless, our interpreter is sufficiently mature to support a broad class of Scheme programs for benchmarking purposes in Section 5. The interpreter itself is written in Common Lisp, using the Common Lisp Object System (CLOS)[2], and the LispWorks[3] development environment. Our code relies on a LispWorks package for multiprocessing and in particular its support for mailbox synchronization.

3.1 A parallel interpreter based on work-stealing

The execution model we assume is based on work-stealing scheduling, in the tradition of multiprocessing languages like Qlisp [21], Multilisp [28], and Cilk [8], with the difference that our interpreter automatically generates parallel tasks instead of requiring the programmer to introduce parallelism. We opt for a work-stealing architecture because this appears to be the most efficient solution to scheduling fine-grained parallelism on multicores [3, 7, 11, 28, 49]. The idea is that we start from a sequential interpreter that we duplicate over as many threads as there are processor cores. Each of these interpreter threads is equipped with a mailbox queue, serving as a *parallel stack*: Interpreter calls are encoded as *messages* put onto the thread's mailbox queue, which the thread continuously polls. We implement a simple work-stealing algorithm so that the different interpreter threads can steal messages from one another to balance the workload.

The code for creating such an interpreter thread is shown in Fig. 3.[4] A Lisp process is initialized through the library function `mp:process-run-function`. The important part of the code is the lambda expression: It encodes what the created process does. As we see, the first thing an interpreter thread does is set up a mailbox queue (`mbox`) and register itself in a global variable that keeps track of all of the interpreter threads (`*interpreters*`). Then an endless loop follows, where the interpreter thread polls its mailbox queue for a message. If a message is found, the message is processed, calling the appropriate interpreter function. If after a particular timeout (here $0.00001s$), no message is found in the thread's own mailbox queue, the thread will attempt to steal a message from one of the other threads (`steal-msg`). The code for stealing a message is omitted. It is just a loop that traverses the list of interpreter threads, polling each mailbox queue until a message is found that can be stolen. This works assuming that a computation is initialized externally by the read-eval-print loop (or REPL), which randomly selects a mailbox queue and assigns it the (first) input expression.

```
(defmethod make-interpreter-thread (name)
  (mp:process-run-function ; create a process
   name '()
   (lambda ()
     (let ((mbox (mp:make-mailbox)))
       (setf (mp:process-mailbox mp:*current-process*) mbox)
       (push mp:*current-process* *interpreters*)
       (do ((msg (mp:mailbox-read mbox :timeout 0.000001)
                 (mp:mailbox-read mbox :timeout 0.000001)))
           (nil)
         (if (not (null msg))
             (funcall (get-function msg) (args msg))
           (let ((msg (steal-msg)))
             (funcall (get-function msg) (args msg)))))))))
```

Figure 3. Code for creating an interpreter thread

Our interpreter is structured as a continuation-passing-style (cps) interpreter [19], i.e. an interpreter where control flow is made explicit using function objects, which makes it easier to distribute computations over different threads. A code excerpt of the evaluator is shown in Fig. 4. An important difference with a regular cps interpreter is that evaluator functions are not directly called, but through the operator `call`, for which the code is also shown in Fig. 4. The operator `call` wraps interpreter calls into message objects and pushes them onto the interpreter thread's mailbox queue.

The interpreter we discussed so far does not give rise to a parallel execution of programs—each interpreter call produces at most one message. We introduce parallelism by revising the interpreter to perform parallel procedure calling and speculative branching.

3.2 Parallel argument evaluation

In our interpreter, all procedures are by default executed in parallel. This includes all user-defined procedures, but also

[1] See `http://people.csail.mit.edu/jaffer/r3rs_toc.html`

[2] The Common Lisp Object System (CLOS) is an extension for object-oriented programming in Common Lisp, see [16] for an overview.

[3] For LispWorks ®, see `http://www.lispworks.com/`

[4] Note that we color Common Lisp code `blue` and Scheme code `green`.

```
(defmethod eval (exp env cont)
    (cond ((variable-p exp) (funcall cont (binding exp env)))
          ((atom exp) (funcall cont exp))
          ((if-p exp) (call #'eval-if exp env cont))
          ((set-p exp) (call #'eval-set exp env cont))
          ...))

(defmethod eval-if (exp env cont) ...)

(defmethod eval-set (exp env cont) ...)
...

(defmethod call (func &rest args)
  (mp:process-send mp:*current-process* (cons func args)))
```

Figure 4. *Calling* the evaluator

primitive procedures and expressions such as `begin`. To execute a procedure in parallel, we evaluate its arguments in parallel. Here we sketch the requirements for going from sequential argument evaluation to parallel argument evaluation. The code for performing sequential argument evaluation is shown in Fig. 5. The function `eval-args` implements a loop that iterates over the list of unevaluated argument expressions (`exps`). The variable `results` is used for accumulating the results from evaluating the arguments. The loop exits when the list of arguments to evaluate is empty, which then triggers the continuation of evaluating the arguments (i.e. calling `cont`, which encodes applying the procedure to the evaluated arguments, which for brevity is omitted from Fig. 5). When there is still an argument to evaluate, the loop calls the top-level evaluator function (`eval`), which it passes as a continuation a lambda that continues the `eval-args` loop with the updated list of evaluated arguments.

```
(defmethod eval-args (exps results env cont)
  (if (null exps)
      (cont results)
    (call #'eval (car exps) env
          (lambda (arg!)
            (call #'eval-args
                  (cdr exps) (cons arg! results) env cont)))))
```

Figure 5. Sequential argument evaluation

Mockup code for evaluating the arguments in parallel is shown in Fig. 6. It illustrates what we would like the code to do. The code implements a loop similar to the loop for evaluating arguments sequentially. The big difference is that the loop for parallel evaluation does not wait for the result of evaluating an argument to proceed with the loop: After requesting the evaluator to evaluate an argument, the loop immediately continues to request evaluation of the rest of the arguments (see the two successive `call`s). As a reminder: `call` transforms an interpreter call into a message object which is pushed onto the local interpreter thread's message queue, from which other interpreter threads can steal messages. It is because of this that the successive `call`s in Fig. 6 are potentially executed in parallel.

The tricky part is that, somehow, calling the continuation of evaluating the arguments (`cont`), which encodes the procedure application, must be delayed until all the arguments

```
(defmethod eval-args (exps results env cont)
  (if (null exps)
      (cont (wait-and-collect-results results))
      (progn
        (call #'eval (car exps) env
              (lambda (arg!) (store! arg! results)))
        (call #'eval-args (cdr exps) env results cont))))
```

Figure 6. Mockup parallel argument evaluation

are finished evaluating and accumulated from the parallel calls (`store!`) in the right order—the order that matches the procedure's formal parameter list.

We conclude that we need two elements to support parallel argument evaluation: 1) a mechanism for accumulating the results of the different arguments, respecting the order of the procedure's formal parameter list and 2) a way of delaying the continuation that encodes the procedure application until the results of all of the argument expressions are known.

3.3 Speculative branching

Here we discuss the requirements for supporting speculative evaluation of conditional expressions. Mockup code is shown in Fig. 7. Under parallel evaluation of a conditional expression, we understand that the test and one of the branches are executed in parallel. The code for `eval-if` shows a `call` to launch the evaluation of the test expression. Then we speculate which branch we are going to execute in parallel (`speculate-alternate-p`), and launch execution of either the then or else expression. We suppose that the results of the different parallel tasks are accumulated in the variables `test`, `then` and `else`. Then, `eval-if` needs to wait for the result of the test expression before it can call the continuation of evaluating the if expression (i.e. `cont`).

```
(defmethod eval-if (exp env cont)
  (let ((test nil) (then nil) (else nil)
        (test-task nil) (else-task nil) (then-task nil))
    (setf test-task
          (call #'eval (car exp) env
                (lambda (test!) (setf test test!))))
    (if (speculate-alternate-p)
        (setf then-task
              (call #'eval (cadr exp) env
                    (lambda (then!) (setf then then!))))
        (setf else-task
              (call #'eval (caddr exp) env
                    (lambda (else!) (setf else else!)))))
    (wait-until-done test-task)
    (if (true-p test)
        (if (not (null then-task))
            (progn
             (wait-until-done then-task)
             (cont then))
            (progn
             (squash else-task)
             (setf then-task
                   (call #'eval (cadr exp) env
                         (lambda (then!) (setf then then!))))
             (wait-until-done then-task)
             (cont then)))
        ...)))
```

Figure 7. Mockup speculative branching

If the test turns out to be true, we verify that we speculated on executing the then branch (by checking that `then-task` is not bound to `nil`). If this is the case, we call the continuation with the result of the then expression, as soon as that result is known. If instead we speculated on evaluating the else branch, which, in this case, is the wrong branch to continue from, we call `squash` to interrupt and discard the else task, and subsequently we start execution of the then task. The code for when the test expression turns out to be false works equivalently and is omitted from Fig. 7.

From this mockup code, we can conclude that what we need to support speculative branching is 1) a mechanism for accumulating the results of the test, then, and else expressions; 2) a way to delay the continuation of the if expression; 3) a mechanism for discarding evaluation of the branch expression that turns out to be the wrong one; 4) a mechanism to figure out which branch to speculate on. Interactions with side effects are discussed in the next subsection.

Note that with speculative branching, stop conditions are bypassed. To illustrate what we mean, consider the definition of a procedure `first`:

```
(define (first l) (if (null? l) '() (car l)))
```

The procedure `first` returns the first element of a list, but if that list is empty, it returns the empty list. When the if expression is evaluated in parallel, and the list happens to be the empty list, it may happen that the evaluator tries to execute `(car l)`, i.e. the else branch. But taking the `car` of the empty list yields an error in Scheme [56]. It is important to note that this error is the result of our parallel evaluation strategy, and we must make sure that such errors are discarded and that they are certainly not reported as programming errors.

Another case where bypassing stop conditions seems paradoxical is that it may cause recursive procedures to recurse beyond the base case. Picture the recursive definition of the factorial function:

```
(define (fac n) (if (<= n 2) n (* n (fac (- n 1)))))
```

Imagine we want to compute the factorial of 2, and that the evaluator has already launched evaluation of the factorial of 1, although the result is just 2. It seems that this execution will loop forever, evaluating the factorial of 1, 0, -1, and so on. We need to make sure that the squashing mechanism of the underlying evaluator is capable of stopping this kind of looping.

3.4 Side effects

One of the key difficulties in parallel programming is handling side effects. This is because side effects impose an explicit ordering in a program's execution, and maintaining this ordering reduces the potential parallelism inherently. To illustrate the problem, consider the following (contrived) procedure for filling a list with numbers from 1 to n:[5]

[5] Recall from Section 3.2 that we parallelize subexpressions of `begin`.

```
(define (make-list n)
  (let ((res '()))
    (define (loop c)
      (if (= c 0) res
          (begin (set! res (cons c res))
                 (loop (- c 1)))))
    (loop n)))
```

Without care, the parallel execution of `make-list` may not produce a list containing numbers from 1 to n: It may produce a list where the numbers are randomly ordered, or even just a subset. This is because it is not fixed when the set! expression is evaluated. When such a situation arises, it is said that a *data race* occurs. Multiprocessing languages provide synchronization constructs for avoiding data races, typically locks or software transactions [29]. Our interpreter should handle synchronization of side effects automatically without the programmer having to mention locks or atomic blocks in the source code.

Mockup code for handling variable assignment automatically is shown in Fig. 8. Before evaluating an assignment, we make sure that all computations that come logically before the assignment have finished executing. That way we can assure that none of those computations uses the (prematurely) updated variable value. For our example above, this means that the assignment waits for the execution of `(= c 0)` and—more importantly—the execution of the iterations 1 to $c-1$. Then, after performing the side effect, we roll back the computations that logically come after the assignment and that (might) read the variable before it was updated. In our example we thus roll back `(loop n)`.

```
(defmethod eval-set! (exp env cont)
  (call #'eval (cadr exp) env
        (lambda (val!)
          (wait-until-all-previous-expressions-done exp)
          (bind! env (car exp) val!)
          (roll-back-all-later-expressions exp)
          (cont 'ok))))
```

Figure 8. Mockup variable assignment

Vector assignment (`vector-set!`) and list assignment (`set-car!` and `set-cdr!`)—after variable assignment the only other primitive side effecting operations in R^{3}RS Scheme—can be handled similarly.

Note that our approach for handling side effects is very conservative—we turn every side effect into a synchronization point. Instead, we could opt for speculative evaluation of side effects and rely on a runtime mechanism to detect and undo data race conflicts. We choose to implement a conservative approach to side effects because we first want to focus on the basic control flow interactions from combining side effects with parallel argument evaluation and speculative branching in a virtual machine. In Section 6, we discuss how alternative approaches to handling side effects could be built on top of our basic support for side effects.

3.5 Summary of requirements

We can generalize and summarize the requirements for adopting parallel argument evaluation and speculative branching as follows:

1. We need a mechanism that accumulates the results of the subcomputations of a computation and then triggers the next continuation.
2. We need a way of delaying the continuation of a computation until all of its subcomputations have finished. (E.g. we can only apply a procedure once we have evaluated all of its arguments.)
3. For implementing speculative branching, we need to make sure that it is possible to discard computations that turn out to be part of the wrong branch.
4. The discard mechanism should avoid the looping that is (seemingly) introduced when parallelizing conditionals bypasses stop conditions in recursive code.
5. We must make sure that the errors produced by bypassing stop conditions are not reported as programming errors.
6. For making sure that side effects happen in the correct order, we need to be able to check whether the expressions that logically come before the side effect have finished computing.
7. We also need a mechanism to roll back computations of expressions that logically come after a side effect.

We argue that all of these requirements are related to managing the control flow, which becomes complicated by dividing the computation over different parallel tasks. In the next section, we discuss the continuator concept for implementing an interpreter that satisfies these requirements.

4. Managing control flow with continuators

We introduce the *continuator* concept as a basis for a runtime mechanism that coordinates the various parallel tasks that are spawned to implement parallel procedure calling and speculative branching.

4.1 Representing the execution trace as a tree

Fig. 9 shows a tree representation of the execution trace of the factorial function. It shows all the subexpressions necessary to evaluate a factorial call. Our interpreter builds this tree as the program runs for tracking the control dependences between computations. We exploit the tree structure as a basis for a runtime mechanism that manages control flow and satisfies the requirements we discussed in Section 3.5.

4.2 Implementing the tree with the continuator

We define the continuator structure for implementing the nodes of the execution tree. For each expression the evaluator processes, there is a corresponding continuator, which is used for *accumulating* and *reducing* the results of (sub)expressions. We implement the continuator as a CLOS class, shown in Fig. 10.

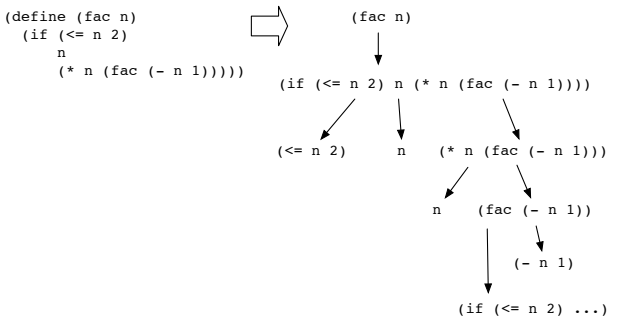

Figure 9. Execution trace of factorial as a tree

```
(defclass continuator ()
  ((arguments :initform (make-array 1))
   (children :initform (make-array 1))
   (parent :initform nil)
   (idx :initform 0)
   (env :initform nil)
   (cont :initform nil)
   (lock :initform (mp:make-lock))
   (msg-handler :initform #'handle-msg)
   (children-exps :initform (make-array 1))))

(defclass procedure-call-continuator (continuator)
  ((fobj :initform nil))

(defclass if-continuator (continuator)
  ((issued-consequent :initform nil)
   (issued-alternate :initform nil)))

(defclass set-continuator (continuator)
  ((var-to-set :initform nil)))
...
```

Figure 10. The continuator structure

The class `continuator` defines a slot `arguments` for storing the results of a continuator's children. The slots `children`, `parent`, and `idx` are there for tracking the parent/child relations of the tree. A continuator also stores interpreter state, in the slots `env` and `cont`. The latter encodes what happens when all of a continuator's children/subexpressions are finished computing. Each continuator has a lock associated with it for synchronization purposes. The slot `children-exps` stores the subexpressions of the continuator, whereas the slot `msg-handler` holds a function. The exact purposes of these different slots becomes clear when we discuss the implementation of parallel argument evaluation, speculative branching, and side effects in the next subsections.

Fig. 10 reveals there are different types of continuators (see the various `continuator` subclasses). There is a continuator class for each of Scheme's primitive expression types, namely procedure call expressions, conditional expressions (`if`), and side effecting expressions (`set!`, `vector-set!`, `set-car!`, and `set-cdr!`). The interpreter dispatches differently on the types of continuators, and some of the specialized continuator classes define slots that hold

additional data, such as a procedure object for the case of the procedure call continuator, slots for checking which branch is executed speculatively for the case of an if continuator, and a slot for storing the name of the variable to assign for the set! continuator.

4.3 Implementing parallel argument evaluation

We extend each evaluator function with a continuator argument. The REPL creates the initial continuator, the root, and subsequent evaluator calls grow the execution tree from there by creating new continuators at well-defined points in the interpreter, namely the interpreter functions that initialize evaluation of a procedure call, or `if`, `set!`, `vector-set!`, `set-car!`, and `set-cdr!` expressions.

Fig. 11 shows the code for parallel argument evaluation, albeit simplified—the synchronization code is omitted (synchronization is discussed in Section 4.6). The function `eval-args` makes a new procedure call continuator (`fcall-ctor`), initializing it using the arguments passed to `eval-args`. There is a check to see whether the list of arguments to evaluate (`args`) is empty: If this is the case, we immediately proceed with the next continuation (`cont`). Otherwise, there is a loop which processes the unevaluated arguments. There are two iteration variables: One for going over the list of unevaluated arguments (`unevaluated-args`) and a counter that gives the index for each of these unevaluated arguments (`ctr`).

```
(defmethod eval-args ((ctor continuator) fobj args env cont)
  (if (null args) (funcall cont '())
    (let ((fcall-ctor
           (make-fcall-continuator ctor cont fobj env args)))
      (do ((ctr 0 (+ 1 ctr))
           (unevaluated-args args (cdr unevaluated-args)))
          ((null unevaluated-args))
        (let* ((arg (car unevaluated-args))
               (ctr* ctr) ; rebind ctr for closure
               (arg-node (svref (children fcall-ctor) ctr)))
          (call
           #'eval arg-node arg env
           (lambda (arg!)
             (call #'fcall-accumulate arg-node arg! ctr*)))))))))
```

Figure 11. Parallel argument evaluation (simplified)

For each unevaluated argument, we request its evaluation (see the call to `eval`). Note the arguments passed to the latter `eval` call, in particular the continuator and continuation arguments. The continuator is the n^{th} child of the procedure call continuator, with $n = ctr$. The continuation calls `fcall-accumulate` with as arguments that node (`arg-node`), the evaluated argument (`arg!`), and the index of the unevaluated argument (`ctr*`). The function `fcall-accumulate` (in Fig. 12) stores the result of evaluating an argument in the procedure call continuator and checks whether all procedure call arguments are evaluated. If so, it calls the continuation that was stored in the procedure call continuator.

We see that the loop in `eval-args` spreads the evaluation of the arguments over different tasks and that their

```
(defmethod fcall-accumulate ((ctor continuator) arg! pos)
  (setf (aref (arguments (parent ctor)) pos) arg!)
  (when (all-arguments-evaluated-p (parent ctor))
    (funcall (cont (parent ctor)) (array->list (parent ctor)))))
```

Figure 12. Accumulating a procedure call argument (simplified)

results are collected and stored in the procedure call continuator (via `fcall-accumulate`). Hence the continuator provides a means to collect the results of parallel computations and satisfies Requirement 1 in Section 3.5 for implementing parallel argument evaluation. Also, we see that the continuation of `eval-args` is delayed using the continuator—it is stored there upon entry of `eval-args` and called only when all of the arguments are finished evaluating (via `fcall-accumulate`). As such, the continuator provides a way of delaying computations and satisfies Requirement 2 in Section 3.5.

4.4 Implementing speculative branching

Fig. 13 shows the code for speculative branching, omitting the synchronization code for conciseness. The code shows that `eval-if` evaluates an if expression by telling the interpreter to evaluate the test (via `issue-test`) and, at the same time, it chooses to evaluate either the else or the then expression (via `issue-alternate` or `issue-consequent`). In our current implementation we always speculate on the else branch, but better speculation strategies, that make guesses based on the test's outcomes in previous executions, are likely more accurate (we come back to this in Section 7.2).

```
(defmethod eval-if ((ctor continuator) exp env cont)
  (let ((if-ctor (make-if-continuator ctor exp env cont))))
    (issue-test if-ctor (cadr exp))
    (if (speculate-alternate-p)
      (progn (setf (issued-alternate-p if-ctor) T)
             (issue-alternate if-ctor (cadddr exp)))
      (progn (setf (issued-consequent-p if-ctor) T)
             (issue-consequent if-ctor (cadddr exp))))))

(defmethod issue-test ((if-ctor if-continuator) exp)
  (let ((child (svref (children if-ctor) 0)))
    (call #'eval child exp (env if-ctor)
          (lambda (test!)
            (call #'accumulate-test child test!)))))

(defmethod accumulate-test ((ctor continuator) val-for-test)
  (let ((if-ctor (parent ctor)))
    (setf (value-for-test if-ctor) val-for-test)
    (if val-for-test
        (progn
          (inline-msg-handler if-ctor 1) ; see squashing
          (when (issued-alternate-p if-ctor)
            (squash-alternate if-ctor)
            (issue-consequent if-ctor))
          (when (not (eql *unknown* (val-for-conseq if-ctor)))
            (funcall (cont if-ctor) (val-for-conseq if-ctor))))
      ...)))
...
```

Figure 13. Speculative branching (simplified)

The code for issuing evaluation of the test expression is also shown in Fig. 13. The function `issue-test` calls `eval` to evaluate the test expression, and the continuation

calls `accumulate-test` (also defined in Fig. 13). This function is responsible for storing the result of the test in the continuator created for evaluating the if expression (`if-ctor`). It then checks if the test result is true or false. If it is true, it calls `inline-msg-handler`, which optimizes squashing, and which we discuss in the next section where we cover the details of squashing computations. After the `inline-msg-handler` call, the code checks if we speculatively started executing the else expression (`issued-alternate-p`), in which case we discard that computation (`squash-alternate`) and issue evaluation of the then expression. Then the function checks if the result of the then expression is already computed and if so, it calls the continuation of the if expression (stored in `if-ctor`). The code for when the test turns out to be false is very similar and omitted from Fig. 13.

The code for issuing evaluation of the then and else expressions (`issue-consequent` and `issue-alternate`) is similar to the definition of `issue-test` and omitted from Fig. 13. The code calls `eval` to evaluate the then (or else) expression, and as a continuation, the function `accumulate-consequent` (or `accumulate-alternate`) is called. This function stores the result of the then (or else) expression in the continuator and if the test result is known and true (or false), the function may call the continuation of the if expression (stored in the continuator `if-ctor`). There is no need for this function to squash the *wrong* branch, as this is handled by `accumulate-test`.

4.4.1 Squashing without stopping the world

Here we discuss the details of discarding (or squashing) computations. In Section 3.5, we discussed that speculative branching may bypass stop conditions and cause evaluation to loop for recursive code; we require that the squashing mechanism avoids that (see Requirements 3 and 4). To refresh our discussion about the looping problem, we discuss Fig. 14, which displays the execution trace of the factorial function. Imagine we want to compute the factorial of 2. Under speculative evaluation, we may speculatively issue evaluation of the else branch of the if in the factorial's definition. However, the result of the factorial is just 2 (i.e the then branch) and the computation of the else branch needs to be discarded.

Fig. 14 shows the execution trace growing to the right, assuming we speculated on the else branch of the if expression. To squash the computation for the else branch, we need to squash its subcomputations, and their subcomputations, and so on. The squashing is illustrated using the red crosses that cut off branches in the execution tree. What may happen is that while we squash a task, e.g. for the expression `(* n (fac (- n 1)))`, one of its parallel subtasks, e.g. for the expression `(fac (- n 1))`, spawns a new subtask. When we come to squashing the task for the expression `(fac (- n 1))`, one of that task's subexpressions (down the tree) may again spawn a new subtask. This may go on forever, given

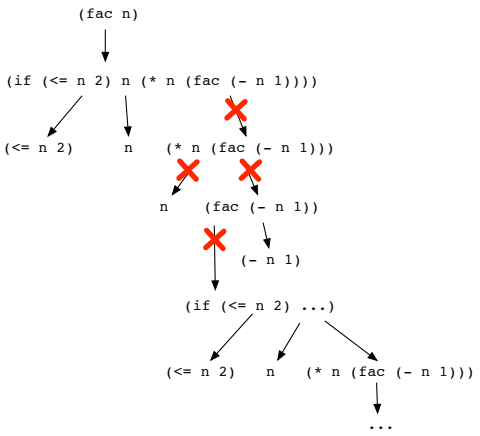

Figure 14. Squashing trying to catch up to task spawning

the recursive structure of the factorial function, so that the squashing is continuously behind the spawning of new tasks. This situation may seem exceptional at a first glance, but it is not. We indeed repeatedly encountered the problem in our first attempts to implement speculative branching.

One strategy to avoid the looping would be to have our squashing mechanism freeze the interpreter to squash all of the tasks that are generated so far and then restart the interpreter. We call this a *stopping the world* approach. It obviously greatly reduces the available parallelism in the execution of a program. Our continuator concept supports a squashing mechanism that does not require stopping the world and still avoids the looping.

Our strategy for implementing the squashing mechanism is to allow the continuators to influence how messages are handled by the interpreter. Recall that interpreter functions are called through an operator `call`, which turns an interpreter call into a message object that is put onto an interpreter thread's mailbox queue, as we explained in Section 3.1 (also see Fig. 4). Such a message consists of the interpreter function to call and its arguments, which for most interpreter functions consists of a continuator, an expression, an environment, and a continuation. We equip each continuator with a *message handler* (a function), and we change the interpreter so that it calls this message handler for processing a message. The idea is that a continuator's message handler can be updated, which allows us to implement the squashing mechanism.

Fig. 15 shows the updated code for creating an interpreter thread—the original code is shown in Fig. 3. The bottom of Fig. 15 also shows the new code for the `call` operator, which now adds a continuator to the message objects it creates. The code for `make-interpreter-thread` shows that interpreter threads no longer (fun)call interpreter functions directly to process a message. Instead, an interpreter thread defers this to the message's message handler (cf. the call to `get-msg-handler`).

```
(defmethod make-interpreter-thread (name)
  (mp:process-run-function ; create a process
   name '()
   (lambda ()
     (let ((mbox (mp:make-mailbox)))
       (setf (mp:process-mailbox mp:*current-process*) mbox)
       (push mp:*current-process* *interpreters*)
       (do ((msg (mp:mailbox-read mbox :timeout 0.000001)
                 (mp:mailbox-read mbox :timeout 0.000001)))
           (nil)
         (if (not (null msg))
             (funcall (get-msg-handler msg) msg)
           (let ((msg (steal-msg)))
             (funcall (get-msg-handler msg) msg))))))))

(defmethod call (f (c continuator) &rest args)
  (mp:process-send mp:*current-process* (cons f (cons c args))))
```

Figure 15. Indirect handling of messages

We define two kinds of message handlers. Their code is shown in Fig. 16. The default message handler (`handle-msg`) just calls the interpreter function stored in the message object. There is also a message handler defined that is used for squashed continuators (`ignore-msg`), which does not do anything with a message but just immediately returns.

```
(defmethod handle-msg (msg) (apply (car msg) (cdr msg)))

(defmethod ignore-msg (msg) t)
```

Figure 16. Two kinds of message handlers

The root continuator has the default message handler. Any other continuator inherits the message handler of its parent, unless it is an if continuator. For if continuators, a new message handler is created that calls the message handler of its parent, as shown in Fig. 17. This allows us to squash children of an if continuator—which can execute speculatively—by replacing their message handlers.

```
(defmethod make-continuator ((pa continuator) idx)
  (let ((node (make-instance 'continuator :parent pa ...)))
    (setf (msg-handler node) (msg-handler pa))
    node))

(defmethod make-continuator ((pa if-continuator) idx)
  (let ((node (make-instance 'continuator :parent pa ...)))
      (setf (msg-handler node)
            (lambda (msg)
              (funcall (msg-handler (parent node)) msg)))
    node))
```

Figure 17. Initializing a continuator's message handler

To squash a continuator, we assign to its parent a dummy continuator that has `ignore-msg` as its message handler, cf. Fig. 18, and we clean up any results that the parent may have stored for the squashed child. Since children tasks that execute speculatively rely on calling their parent's message handler, they are automatically squashed as well. Note that squashing a continuator does not mean that all of the messages produced for its children are discarded at exactly the same time. These messages are only discarded as soon as an interpreter thread tries to process them. However, this way of squashing does not require us to freeze the interpreter and stop the world!

```
(defmethod squash-continuator ((ctor continuator))
  (let ((old-pa (parent ctor)))
    (setf (parent ctor)
          (make-instance 'continuator :msg-handler #'ignore-msg))
    (setf (svref (children old-pa) (idx ctor))
          (make-instance 'continuator :pa old-pa :idx (idx ctor)))
    (setf (svref (arguments old-pa) (idx ctor)) *unknown*)))
```

Figure 18. Squashing a continuator

At this point, one may wonder about the performance impact of redirecting message handing through message handlers. Fig. 17 suggests we create a new indirection for each child of an if continuator. When executing recursive code, this can lead to very deep chains of indirections in the message handlers. For example, consider we want to evaluate the factorial of 35: each time we get to evaluating the if expression, we create an indirected message handler. This means that each recursive call—of which there are a lot for computing `(fac 35)`—takes place through an indirected message handler. We can however undo the indirections for the children of an if continuator as soon as we know the outcome of the test. It is then safe to undo the indirection for the correct branch since that branch cannot be squashed anymore. Hence we replace that continuator's message handler with the one of the if continuator. This is what `inline-msg-handler` does in the code for `eval-if` (in Fig. 13).

4.4.2 Discarding errors caused by speculation

With speculative branching, stop conditions may be bypassed. In Section 3.3, we discussed an example where this causes the execution of a program to generate unexpected errors. We discussed the procedure `first` which takes as argument a list and returns the `car` of that list, unless that list is empty, in which case it returns the empty list. Under speculative execution, we may try to take the `car` of the empty list which in Scheme is an error. Such errors are created by our parallel evaluation strategy and we do not want them to propagate as programming errors (see Requirement 5 in Section 3.5).

Normally, when an error occurs, execution is halted and the error is spit out to the programmer. In our parallel interpreter, we change this so that errors are propagated as results of an execution. Fig. 19 shows the code for resolving a procedure call. The case for applying a primitive procedure, i.e. a procedure that is defined in Common Lisp such as `+`, is wrapped with a `handler-case`[6]. We see that when a primitive procedure application causes an error, we just call the continuation anyway, with the error object as result. This way, errors that are generated by speculation do not interrupt the execution, but are propagated as results of the wrongly speculated branches. Since those branches are never returned

[6] `handler-case` is the equivalent to `try-catch` in other languages.

as the result of the computation as a whole, these errors remain hidden from the programmer.

```
(defmethod procedure-apply ((ctor continuator) fobj args env cont)
  (let ((fcall-ctor (parent ctor)))
    (if (primitivep fobj)
        (handler-case
            (funcall cont (apply (primitive-f fobj) args))
          (error (condition) (funcall cont condition)))
      (call #'eval fcall-ctor (body fobj)
            (bind (func-env fobj) (params fobj) args) cont))))
```

Figure 19. Applying a procedure (simplified)

4.5 Implementing side effects

We now discuss the implementation of side effects in our parallel interpreter. We discuss only the implementation of variable assignment (`set!`), because the implementations for vector and list assignment are practically the same.

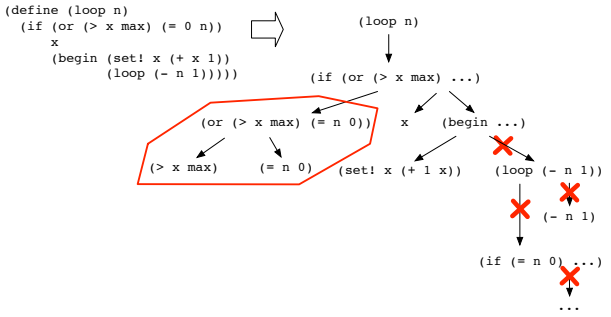

Figure 20. Waiting and rollbacks for side effects

To refresh our discussion on side effects, consider the (contrived) loop example in Fig. 20.[7] This loop iterates n times, and during each iteration, it increases the value of the variable x by 1, as long as the value of x is smaller than max. There is one side effect in the loop, the assignment to x. The (sequential) semantics of the loop dictate that this assignment occurs *after* the check to see if x is bigger than max and *before* iteration $n-1$. However, the implementation of variable assignment in our parallel evaluation strategy must explicitly make sure this is the case.

Our implementation exploits the continuator tree to find out when it is safe to perform a side effect and also to figure out which computations were started prematurely and need to roll back. Consider the continuator tree drawn for our running example in Fig. 20. We know we can perform the assignment when the computations/continuators to the left of the set! continuator completed. We can check this by following the parent pointers of the continuator for the `set!` and verifying that all of their children to the left have already reported their results (Requirement 6 for implementing side effects in Section 3.5). Similarly, we know what computations potentially need to roll back: It is the computations/continuators to the right of the set! continuator that potentially read the *old* value of x and need to be retried (Requirement 7 in Section 3.5).

[7] Recall from Section 3.2 that we parallelize subexpressions of `begin`.

```
(defmethod eval-set ((ctor continuator) exp env cont)
  (let ((set-ctor (make-set-continuator ctor exp env cont)))
    (let ((child (svref (children set-ctor) 0)))
      (call #'eval child (caddr exp) env
            (lambda (exp!)
              (call #'accumulate-set child exp!))))))

(defmethod accumulate-set ((ctor continuator) val)
  (if (expressions-issued-before-done-p ctor)
      (let ((set-ctor (parent ctor)))
        (set-binding (env set-ctor) (var-to-set set-ctor) val)
        (rollback-expressions-issued-after set-ctor)
        (update-argument set-ctor val 0)
        (funcall (cont set-ctor) 'ok))
    (call #'accumulate-set ctor val)))
```

Figure 21. Variable assignment (simplified)

Fig. 21 shows the implementation of variable assignment. The method `eval-set` creates a set! continuator and calls `eval` for computing the variable's new value. The interesting part is the continuation which calls `accumulate-set`. This function loops until it finds out that all expressions that were issued for evaluation before the set! expression finished (via the function `expressions-issued-before-done-p`). The code for `expressions-issued-before-done-p` is omitted, but it is a loop that follows the parent pointers starting from the set! continuator (`set-ctor`) and checks the results these continuators have accumulated so far. The function `expressions-issued-before-done-p` has two cases, depending on whether it checks an if continuator or another type of continuator. By default, the children *to the left* of a particular continuator are all of its parent's children that have a smaller index. In case of an if continuator, however, the only child that is considered to be *to the left* of the continuator for the else branch is the continuator for the test, excluding the continuator for the then branch. This is because either the then or the else branch is selected in the end, and there is no point for an else computation to wait for a (squashed) then computation!

When all of the expressions issued before the assignment are done, `accumulate-set` performs the actual assignment by overriding the value of the variable (`set-binding`). Then the expressions issued after the set! expression are rolled back (via `rollback-expressions-issued-after`). The code for that is omitted: It is just a loop that follows the set! continuator's parent pointers, squashes those continuator's children to the right, and reissues them, by calling `eval` with the expressions that they stored in the slot `children-exps` (see the definition of the `continuator` class in Fig. 10). Note that these rollbacks do not need to undo any assignments because each assignment is a synchronization point that waits for all of the expressions that are issued before. The squashing mechanism used to perform rollbacks is similar to what we defined in the previous subsection for if expressions. The latter means that code with

side effects requires all continuators to have an indirected message handler by default.

4.6 Summary and implementation pitfalls

In this section, we have introduced the continuator concept and have shown how it helps coordinating parallel argument evaluation, speculative branching, and side effects. All the requirements listed in Section 3.5 are fulfilled:

- Requirements 1 and 2 (accumulating arguments that are evaluated in parallel, and delaying the continuation of a procedure call) are fulfilled by the procedure call continuator, as described in Section 4.3.
- Requirements 3 and 4 (the ability to discard wrongly speculated branches, without looping) are fulfilled by the message handler architecture, as described in Section 4.4.1.
- Requirement 5 (avoiding reports of errors that are thrown in wrongly speculated branches) is fulfilled by wrapping errors and returning them as results that will eventually be discarded, as described in Section 4.4.2.
- Requirements 6 and 7 (ensuring correct order of side effects, and rolling back computations that depend on side effects) are fulfilled by the set! continuator, as described in Section 4.5.

The code we presented is simplified for explanation purposes. There are, however, a number of implementation difficulties that we want to make the reader aware of:

- Continuator access must be synchronized, and we follow a strict locking protocol to avoid deadlocks (first we lock the parent, then the continuator itself), which pollutes our implementation with checks that the continuator we locked was not squashed after we locked its parent and that its parent is really the same as the one we locked.
- When rolling back expressions for implementing side effects, we must first squash *all* expressions issued after the side effect, and only then restart them. Otherwise we might squash and restart a continuator before those other expressions are squashed. Those may have remaining (but incorrect) results, and the restarted continuator may trigger these continuators to complete.
- We need to create the children of a continuator right away when we create the continuator itself, even though we do not know what type of continuators these children should be. This is because they can be squashed *before* they are used for evaluation. Because of this, there are certain places in the interpreter that cast continuators to the right type (`eval-fcall`, `eval-if`, `eval-set!` and the interpreter functions for the other side effects).
- For interpreting the application of a non-primitive procedure, the interpreter reuses the continuator that was generated for evaluating the expression that calls the procedure. E.g. the procedure call continuator for `(fac (- n 1))` is reused as continuator for evaluating its expansion into `(if (<= n 2) 2 (* n ...))`. Otherwise there would be a hole in the continuator tree.

5. Validation

We ran our parallel interpreter on the Clinger benchmark suite for Scheme [14]. Although our interpreter is a proof of concept rather than an industrial-strength implementation of the ideas presented in this paper, the benchmarks already show that the overhead of the continuator infrastructure is in an acceptable range and good speedups are possible.

The Clinger benchmark suite for Scheme is a collection of different benchmark suites, including the Gabriel benchmarks [20], the Kernighan and Van Wyck benchmarks [38], and a set of numerical and other programs. The applications include simple programs like the Fibonnaci and Ackermann functions, but also AI algorithms for solving puzzles, numerical programs for computing fast Fourier transformations and Mandelbrot sets, a type checker, and so on. Our implementation runs all of the programs, except for the ones that rely on `call/cc`, which we do not support, as it is not clear what a continuation represents in our parallel interpreter. We also do not support the benchmarks for testing file I/O. We ran our experiments using 32-bit LispWorks 6.0 for Mac OS X, on an 8-core Mac Pro, equipped with two Quad-Core Intel Xeon "Nehalem" processors (each 2.26GHz) and 8GB of 1066MHz DDR3 ECC SDRAM.

In our benchmarks, we compare two interpreters, namely our parallel interpreter and a sequential cps interpreter. These two interpreters only differ in the implementation of the evaluation process. They share all of the code that implements the interpreter infrastructure, such as the code for representing environments, parser code, code for representing Scheme values, and so on. Our benchmarks measure two things: 1) the memory overhead generated by the continuator infrastructure and 2) the speedups of the different programs.

Fig. 22 shows the benchmark results when a straightforward implementation of a Scheme interpreter and continuators is used. It shows the speedup per benchmark using 2, 4, 6, and 8 threads. The graph maps the number of threads (X axis) onto the speedup factor (Y axis), which is computed as $\frac{T_{seq}}{T_{par}}$, where T_{seq} is the runtime of a benchmark using the sequential interpreter, and T_{par} is the runtime using the parallel interpreter. Fig. 22 reveals there are benchmarks that show speedups, but also benchmarks that show slowdowns, as their speedup factor is below 1. Using 8 threads, *nqueens* is about 1.6x faster, *tak* 1.8x, *takl* 3.5x, and *fib* 1.3x. Additionally, these speedups scale with the number of cores. Programs that contain side effects (see *paraffins!*, *fft!*, and *mbrot!*, i.e. all benchmarks with a *!* in their name) run $2-90$x slower, but roughly in constant time, independent of the number of interpreter threads. That is not so surprising, as our implementation insists on sequentializing side effects,

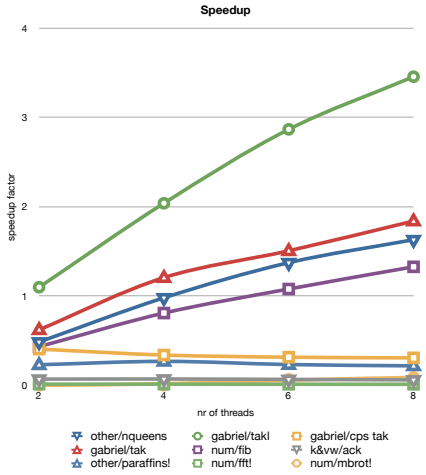

Figure 22. Speedups and slowdowns

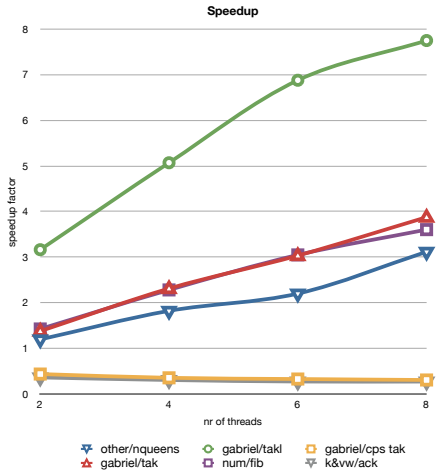

Figure 23. Speedups (when optimized for functional code)

and consequently there is little parallelism to be exploited in programs with many side effects. There are some opportunities to improve the way side effects are handled and we discuss them in Section 6.

Oddly enough, there are also some benchmark programs without side effects that run slower, namely *ack*, an implementation of the Ackermann function, and *cps tak*, a continuation-passing-style implementation of the Takeuchi function (see [20]). If we look at the execution traces of these programs, we observe that they do not produce balanced trees, but rather unbalanced trees with linear structure. The nodes of these trees accumulate their results one by one, and consequently there is not much parallelism in the computations. This is, for example, also the case for the factorial function, for which we display the execution trace in Fig. 9 in Section 4.1. Computing a factorial yields a chain of nested factorial calls—which grows until the base case is reached—and then the actual result is computed by returning from the recursive calls and multiplying their results, one by one. The latter is inherently a sequential process, and such code cannot benefit much from parallelization. In contrast, if we look at the execution traces of the programs in Fig. 22 that yield a good speedup, we observe that these programs produce more balanced execution trees, and this is because they have (some) important parts written in a tree-recursive style. In other words, programs written in a tree-recursive style are more likely to produce good speedups. However, programming in a tree-recursive style is not enforced by Scheme, which makes it difficult to parallelize existing programs.[8]

We can greatly improve the benchmark results we just discussed by applying an optimization that targets functional code. A simple static analysis of the code can reveal the absence of side effects which allows us to remove a great deal of indirections in the continuator's message handlers. Fig. 23 shows the speedups that we get by using this optimization. We see that the speedups are improved by at least a factor 2 compared to Fig. 22. Using 8 threads, *nqueens* is now 3.1x faster, *tak* 3.9x, *takl* 7.8x, and *fib* 3.6x. Note that for *takl* we even get a super linear speedup when using 2, 4, or 6 threads. The optimization does not remove the overall slowdowns for *ack*, and *cps tak*, nor does it help for code with lots of side effects (not shown).

Fig. 24 shows the impact of the continuator infrastructure on memory usage. It shows the memory overhead per benchmark using 2, 4, 6, and 8 threads. The graph maps the number of threads (X axis) onto the overhead factor (Y axis), which is computed as $\frac{M_{par}}{M_{seq}}$ where M_{seq} is the total amount of memory consumed using the sequential interpreter, and M_{par} is the total amount of memory consumed using the parallel interpreter. The figure reveals for *nqueens*, *tak*, *takl*, and *fib*, that parallel execution consumes 2 to 3 times more memory than sequential execution. The figure also shows that for these cases, the memory overhead grows very slowly and is almost independent from the number of interpreter threads. The small increase of memory usage is because when there are more interpreter threads, more of the *wrongly* speculated computations can execute, which creates unnecessary continuator and message objects. However, since the speedup for *nqueens*, *tak*, *takl*, and *fib* also increases with the number of threads (i.e. the results are computed faster), this effect is negligible. In contrast, the memory overhead for *ack* and *cps tak* grows more steeply with the number of threads (see Fig. 24). When discussing the speedups, we said that these programs currently do not give a speedup because their linear structure forces part of the computation to proceed sequentially. Because of this, the effect on memory overhead

[8] It is interesting to compare this to Fortress, which is a new multiprocessing language that is currently being developed at Sun Labs, and is designed to encourage programming in a tree-recursive style. We discuss Fortress in Section 7.5.

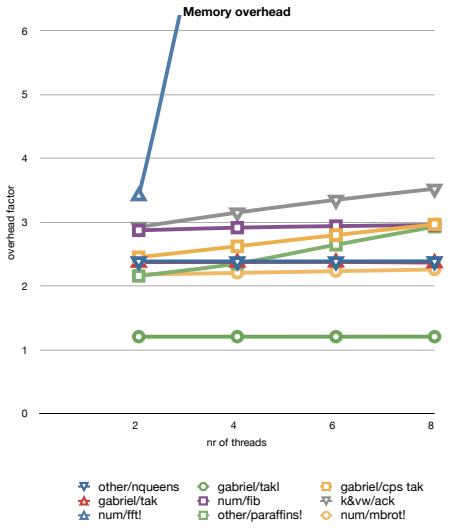

Figure 24. Memory overhead

for speculating wrongly is more noticeable for *ack* and *cps tak*. In future work, we plan to investigate better branch prediction techniques: This should reduce the number of wrong speculations, and consequently also the extra memory overhead that it generates.

Memory usage is very different for the benchmarks with side effects (*fft!*, *paraffins!*, *mbrot!*), where memory usage grows fast with the number of threads. This is because our interpreter synchronizes on every side effect and restarts all computations that come logically after the side effect, and these restarted computations need to create new continuator objects. Also, the more threads there are available, the further the computations that need to restart may have progressed, which may have created plenty of continuator objects that are then discarded. Note that the memory overhead for *fft!* grows spectacularly, and its graph is even cut off to keep Fig. 24 readable. This is because the *fft!* program contains *exceptionally* many side effects, as it defines a huge number of variables which it initializes using assignments.

To summarize: 1) There are some programs for which we get (linear) speedups, 2) the memory overhead of the continuator infrastructure is a constant factor for these programs, and 3) programs with side effects run substantially slower using our approach, but this is due to our naive implementation for handling side effects and not because of the overhead of the continuator infrastructure. Given these observations, we conclude that the overhead of the continuator infrastructure can be minimized and is in an acceptable range, but also that there is plenty of room for optimizations to realize a proper implementation of our approach.

6. Outlook: Integration into virtual machines

Apart from the fact that our interpreter automatically parallelizes Scheme programs, it is otherwise a very traditional interpreter that does not use any of the known techniques to speed up programs by way of just-in-time or dynamic compilation. The reason is mainly because our goal was to focus our attention on understanding how automatic parallelization of recursive code could work, and studying a parallel interpreter helped us discover the continuator concept.

Parallelizing an interpreter alone is, of course, not enough. One approach to reusing our ideas would be in a compiler that generates code to spawn concurrent tasks for argument expressions in procedure calls, and speculative branching in conditional statements. Continuators can then be introduced by the compiler as well to track and manage the runtime dependences, just as described in this paper. We are convinced that reusing the continuator concept in a compilation setting should be straightforward, since we were already able to reuse the continuator concept unchanged in a non-trivial variation of our interpreter that is organized as a pipeline (not discussed in this paper). Integration with a compiler can have the advantage that static analysis techniques could be used to determine which side effects are strictly local (such as local assignments to initialize freshly allocated data structures, or local assignments to iteration variables that are otherwise not accessed). See [4, 37] for an overview of such analysis techniques.

However, we actually believe that the ideas of this paper could more effectively be used in a dynamic compilation setting: Virtual machine implementations like HotSpot for Java [41], V8 [26] and TraceMonkey [23] for JavaScript, and LuaJIT for Lua [48], to name just a few, actually initially execute code in interpreted mode, but monitor the interpretation to discover "hot" regions of commonly executed code or code paths, which can then be compiled at runtime to speed up overall execution of a program. The idea is that most of the time, only small parts of a program are executed, and that dynamic compilation can then spend its resources on optimizing the identified hot regions. For such optimizations, the dynamic compiler can then take runtime information into account that was gathered by the virtual machine and that would otherwise not be available to a purely static compiler. (See also [6] for a historical overview of just-in-time compilation.)

Our hypothesis is that automatic parallelization of recursive code and continuators, as presented in this paper, could be integrated into such a virtual machine architecture. On the one hand, the static analysis techniques for distinguishing between local and global side effects could be reused when compiling the hot code regions to reduce the observed negative impact of assignments by forcing such code to execute sequentially.

On the other hand, the virtual machine could perform further analyses at runtime: For example, it could measure both

sequential and parallel execution of the same code regions, and subsequently run only those regions in parallel that actually benefit from such parallelization. The virtual machine could also attempt to limit the amount of parallelism that is generated, by parallelizing argument expressions and conditional statements only to a certain depth in the computation tree, again steered by previously monitored runtime information (as already done in Qlisp [25, 49, 65]). In other words, the virtual machine could perform a variant of auto-tuning at runtime. To summarize, we believe that the notion of automatically parallelizing recursive code using continuators opens up a number of exciting avenues for future virtual machine research.

7. Related work and further discussion

7.1 Parallelizing compilers

Automatic parallelization has been a major research topic since the 1960's [37]. The focus of that research is finding new compilation techniques, mostly in the context of Fortran. Note that the term *automatic parallelization* is used for two different kinds of compiler research (see Chapter 1 of [37]). It is used for referring to compilers that optimize code so that it better exploits the parallel design of pipelined processors, but the term is also used for referring to compilation techniques that parallelize programs at a coarser grain by mapping code onto threads to run on multicore processors. Our own work is similar to the latter kind of research as we want to parallelize sequential programs so that they can better benefit from multicore processors.

We observe that the main sources of parallelism that is considered in the literature are *loops*, as it is in principle possible to map the different iterations of a loop onto different threads. The major difficulty is that, in order to preserve the sequential semantics, the execution of these threads must respect the *data dependences* between the iterations. That is, the execution must respect the order of the side effects. Some approaches allow parallelizing only loops for which it can be statically determined that there are no dependences between the iterations that may cause a faulty parallel execution. The focus of these approaches is finding better static analysis techniques to improve discovery of dependences [37], techniques for rearranging code to improve data locality [9], and finding ways for transforming loops so there are less dependences between iterations [4, 10]. In contrast, *software speculation* approaches allow parallelizing arbitrary loops, but they foresee runtime mechanisms for detecting and undoing dependence violations. Current research on software speculation focuses on finding ways to reduce the memory overhead associated with runtime dependence checking, either by finding better dependence checking algorithms and implementations [54], or by reducing the number of threads that are generated or activated at the same time [12, 52].

In our work, we focus on automatically parallelizing recursive code, rather than loops. For parallelizing loops, the iterations are mapped onto threads, but recursive code requires a different way of extracting parallelism. Work on parallel programming languages suggests constructs for parallel procedure calling and speculative computation. We propose a virtual machine that combines parallel procedure calling and speculative branching as default execution strategies. Our contribution is the continuator concept for managing the control flow interactions between parallel argument evaluation, speculative branching and side effects.

However, similar to the case of loops, we also require a mechanism for handling data dependences between the parallel tasks. Our continuator infrastructure allows hooking in such a mechanism, but as we discussed, we have only tried a conservative way of handling data dependences (every side effect is a synchronization point). It would be interesting to reuse the ideas for managing data dependences in loop parallelization. We could for example add a preprocessing step to our virtual machine that performs a static analysis of the code to detect data dependences, e.g. to mark (local) variables that can be safely updated in parallel.

Perhaps one obstacle is that most of the existing dependence analyses focus on statically allocated arrays, but recursive code often operates on more complex data structures. There is more advanced work by Rugina and Rinard [53] on dependence analysis in the context of divide and conquer algorithms where the focus is on dynamically allocated arrays. An alternative strategy for handling side effects would be to opt for a runtime mechanism for handling dependence violations as used in loop parallelization based on software speculation [12], or reuse previous work on software transactions [15, 32, 42].

7.2 Processor parallelism

Exploiting parallelism has always been a key optimization strategy in processor design. In fact, it is the key idea behind pipelined and multiple issue processors, which, before multicores came along, dominated the desktop market [30]. Pipelined processors parallelize the execution of a program as it runs, as their components (or *stages*) are organized in such a way that subsequent instructions are executed in an overlapping manner. Various updates of the basic pipelined design is what steadily improved the performance of uniprocessors for the past twenty years. However, due to various technical limitations, involving heat production and power consumption, hardware manufacturers can no longer improve upon the current pipelined designs in a cost-effective way [30], and instead focus shifted to multicores. The big difference with previous designs is that multicores do not automatically make use of the available parallelism. That is, the hardware is not responsible for automatically distributing program execution over the different cores. In multicore designs, that is the software's responsibility.

Our idea is to transfer the notion of runtime parallelization in hardware to the virtual machine. The difference is that our virtual machine operates on a coarser tree-based

program representation than the pipelined hardware, which operates on flat machine code. This tree-based program representation is the basis for spreading execution over different processors, but it also introduces new complexity.

The major idea from hardware parallelization that we explore in this paper is *speculative branching*. A branch instruction in machine code consists of a test instruction and a jump instruction that tells the processor what to execute next when the test instruction is true. This means the processor has to execute the test instruction before it can select the next instruction to execute. Because of this, the pipeline's use is suboptimal, as the test instruction needs to be *completely* done before the next instruction can start executing. A processor with speculative branching speculates on the next instruction, e.g. the one from the branch, to keep the pipeline busy. When the test is known and it turns out the processor speculated wrongly, the speculated instruction is squashed by clearing everything in the pipeline that is behind the test instruction. The hardware often implements *branch prediction* to speculate on the instruction that most likely comes next, e.g. by making guesses based on the outcomes of the test for previous executions.

In this paper, we propose to execute high-level conditional constructs speculatively. The major difficulty is that the squashing mechanism becomes more complicated. In the case of the pipelined processor, it is easier to find the computations that need to be squashed, as it is just whatever is in the pipeline right after the test instruction. Also, machine code is flat, and it is known for each instruction what registers it accesses and hence what data dependences exist between instructions. In contrast, in our system, the execution of conditionals is scattered over different concurrent processors and it is not known what variables, arrays, etc. an expression is going to access. Our contribution is that we explore speculative branching in the context of high-level language constructs and that we show how squashing can be supported using the continuator concept. We are aware of a couple of other software systems that support speculative branching for high-level conditionals [45, 47], but none of these approaches correctly handles squashing of wrongly speculated computations (see Section 7.4). We have not yet explored branch prediction in our system and leave this as a topic for future work.

7.3 Multiprocessing languages

An alternative to automatic parallelization is explicit parallel programming, as supported by multiprocessing languages such as Qlisp [21], Multilisp [28], and Cilk [8]. Qlisp and Multilisp are Lisp dialects, whereas Cilk is an extension of C. Cilk introduces a `spawn` construct for launching parallel tasks and a construct called `sync` for marking where the code needs to wait for those tasks to finish. Qlisp introduces the `qlambda` construct for creating functions whose bodies are evaluated as parallel tasks, a construct `pcall` for calling a function whose argument expressions are processed in parallel, and a synchronization construct called `wait`. Multilisp also has the `pcall` construct and introduces the `future` construct for evaluating an expression in a new thread. The return value of such a `future` call is a special *future object* to which the spawned thread (eventually) writes back its result. The idea is that the execution can proceed from a `future` call, using the future object as the `future` call's return value. When the Multilisp implementation tries to apply a primitive function with such a future object, it is supposed to use the result that is written back to the future object, and the implementation blocks the primitive application until this is the case. Hence in Multilisp, no explicit task synchronization is necessary.

In our approach, the virtual machine is responsible for choosing where to spawn parallel tasks. It performs parallel argument evaluation and speculative branching by default, and the continuator infrastructure is responsible for synchronizing the spawned tasks. In other words, the language constructs from Qlisp, Multilisp, and Cilk are absorbed by the virtual machine.

There seems to be a strong relation between futures and continuators. Future objects collect the result of a parallel task, and they force function applications to block until the result is known. Similarly, continuators accumulate the results of different parallel tasks and delay their continuation. The difference between future objects and continuators is that the future object introduces a single point of synchronization in the implementation, namely for primitive function application, whereas the continuator infrastructure introduces multiple synchronization points, namely for evaluation of procedure calls, `if`, `set!`, `vector-set!`, `set-car!` and `set-cdr!` expressions. The continuator is designed for implementing automatic parallel argument evaluation and speculative branching, and it is not clear if this can be done using future objects alone.

A major contribution of multiprocessing languages like Qlisp, Multilisp, and Cilk is that they introduced and developed very advanced work-stealing schedulers [7, 8, 21, 28, 49], which are being picked up by modern multiprocessing languages such as Fortress [11] and X10 [3]. Our interpreter similarly implements a runtime architecture based on work-stealing. The implementation itself is, however, a straightforward one. It would be interesting to run our benchmarks with one of these more advanced work-stealing schedulers, where data structures, synchronization points, and scheduling strategies are heavily optimized [2, 3, 7, 8, 11].

7.4 Speculative computation

Speculative computation is a parallelization strategy where computations are executed before they are known to be required [45, 47, 50]. It has been implemented in Multilisp and Qlisp, where it appears in the form of parallel variants of constructs like `funcall`, `and`, `or`, and `if`. Experiments with speculative computation are promising: For example, for certain Multilisp applications, a speedup factor of 26 was

reported [47]. However, speculative computation is very difficult to implement, and it is currently not supported by any production language.

There are a couple of problems with the implementation of speculative computation in Multilisp. In particular, a correct implementation of discarding threads does not seem realized (see Chapter 10 of [46]), and discarding of threads may lag behind new threads being spawned. In [46], the author argues that this should not occur in concrete executions because, in the existing implementation of Multilisp, the concrete time to spawn a new thread is much higher than the time for aborting a thread. As such, the problem should not occur. This is a shaky reasoning at best, and not a solution for implementing speculative computation correctly.

A similar story can be found in a paper that reports on the parallel implementation of a rule-based production system in Qlisp [45]. The authors claim that speculative evaluation is beneficial for this application, and propose to add a speculative construct `sfcall` to Qlisp for speculatively executing a function. However, they acknowledge they were unable to implement a mechanism to correctly discard wrongly speculated computations. It seems that a correct implementation for correctly discarding wrongly speculative computations is lacking in existing systems [45, 47, 50].

In the virtual machine we propose, conditional expressions are by default executed speculatively. We did not investigate speculative execution of other types of expressions, but we are confident that they can be supported by the continuator infrastructure as well (since `or` and `and` can be macroexpanded into `if` [56]). Also, our implementation based on continuators and message handlers makes it possible to squash wrongly speculated computations without freezing the execution *and* guarantees that wrongly speculated computations do not keep spawning new tasks. We think this a fundamental contribution.

7.5 Fortress, an implicit parallel language

Fortress [5] is a new programming language that is designed for high-performance computing and is currently being developed at Sun Labs. What sets Fortress apart from other multiprocessing languages is that it is an *implicit parallel language* in which the programmer does not need to introduce parallelism explicitly. Instead, the semantics of Fortress are defined so that the arguments of an operator *may* execute in parallel [60]. The programmer must take these parallel semantics into account when writing programs, e.g. side effects must be synchronized explicitly using atomic blocks.

One of the key ideas behind Fortress is to replace the basic linear data structures (i.e. vectors and lists) with the *conc* data structure, which organizes data as a tree. Iteration code for the *conc* is written in a tree-recursive style, and the claim is that such code benefits more from parallelization than iteration code that operates on linear data structures [60]. Our own experiments confirm this observation, as our benchmarks show the best results for programs where important parts are written in a tree-recursive style.

There is obviously an overlap between Fortress and our own work, albeit our starting points are different. Whereas Fortress's goal is to develop a new multiprocessing language in which it is easy to write code that is optimal for parallel execution, our goal is to find out how we can automatically parallelize existing recursive programs. Similar to our approach, Fortress uses parallel argument evaluation as one means to introduce parallelism in a program's execution. However, on top of that, we provide speculative evaluation of conditional expressions, and we sequentialize the behavior of side effects.

8. Summary and conclusions

In this paper we explore the idea of a work-stealing virtual machine that automatically parallelizes recursive code by combining parallel procedure calling and speculative execution of conditionals as default execution strategies. Our contribution is the continuator concept for managing the control flow interactions between parallel procedure calling, speculative branching, and side effects. The idea is to represent the execution trace as a tree where we use continuators to represent its nodes. The continuators are used for accumulating and reducing the results of parallel computations. Also, they guarantee discarding computations issued by wrong speculation without having to freeze the interpreter, and they make rollbacks possible for handling side effects.

As a proof of concept implementation, we present a parallel Scheme interpreter that implements our approach. We choose Scheme because it is a minimal language in which it is possible to write anywhere on the spectrum of very functional code to very imperative code. As a validation, we show the results of our interpreter running the Clinger benchmark suite for Scheme. The numbers show (linear) speedups for functional code where important parts are written in a tree-recursive style. Programs with side effects in general run slower, but in a constant execution time independent from the number of available processor cores.

Our hypothesis is that automatic parallelization of recursive code and continuators, as presented in this paper, could be integrated into a full-fledged virtual machine architecture. In future work, we plan to move our implementation to a virtual machine that performs just-in-time and dynamic compilation. In such an architecture, we could reuse existing compilation techniques for improving the way side effects are handled, e.g. by adding a static analysis step to detect side effects that can execute in parallel, or by deciding to execute highly imperative code sequentially. Also, the virtual machine could perform further analyses at runtime, and it could measure both sequential and parallel execution of the same code regions, and subsequently only run those regions in parallel that actually benefit from such parallelization.

Acknowledgments

We thank our shepherd Richard P. Gabriel for his support and advice to improve this paper. A very big thank you goes to Dave Fox, Usha Millar, and Martin Simmons from LispWorks® for letting us use an alpha version of LispWork 6.0 since early 2008. Our work would not have been possible without LispWorks 6.0 and its support for multiprocessing. It is truly an amazing piece of technology that sets a very high standard for multiprocessing support in Lisp and other dynamic languages. We thank the participants of the POOSC'08 workshop (at ECOOP'08) where our initially vague ideas were warmly received and discussed. We also thank Theo D'Hondt, Wolfgang De Meuter, Guy L. Steele Jr., Luc Steels, and Stijn Verhaegen for their comments on earlier drafts of this paper and other discussions on this work. Charlotte Herzeel's research is funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

References

[1] H. Abelson and G. J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, MA, USA, 1996. ISBN 0262011530.

[2] U. A. Acar, G. E. Blelloch, and R. D. Blumofe. The data locality of work stealing. In *SPAA '00: Proceedings of the twelfth annual ACM symposium on Parallel algorithms and architectures*, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-185-2. doi: http://doi.acm.org/10.1145/341800.341801.

[3] S. Agarwal, R. Barik, D. Bonachea, V. Sarkar, R. K. Shyamasundar, and K. Yelick. Deadlock-free scheduling of X10 computations with bounded resources. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 229–240, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-667-7. doi: http://doi.acm.org/10.1145/1248377.1248416.

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools with Gradiance*. Addison-Wesley Publishing Company, USA, 2007. ISBN 0321547985, 9780321547989.

[5] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. Steele, Jr., and S. Tobin-Hochstadt. The Fortress Language Specification, version 1.0, March 2008.

[6] J. Aycock. A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113, 2003. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/857076.857077.

[7] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46(5):720–748, 1999. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/324133.324234.

[8] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *SIGPLAN Not.*, 30(8):207–216, 1995. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/209937.209958.

[9] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. *SIGPLAN Not.*, 43(6):101–113, 2008. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1379022.1375595.

[10] P. Boulet, A. Darte, G.-A. Silber, and F. Vivien. Loop parallelization algorithms: from parallelism extraction to code generation. *Parallel Comput.*, 24(3-4):421–444, 1998. ISSN 0167-8191. doi: http://dx.doi.org/10.1016/S0167-8191(98)00020-9.

[11] D. Chase and Y. Lev. Dynamic circular work-stealing deque. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28, New York, NY, USA, 2005. ACM. ISBN 1-58113-986-1. doi: http://doi.acm.org/10.1145/1073970.1073974.

[12] M. Cintra and D. R. Llanos. Toward efficient and robust software speculative parallelization on multiprocessors. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 13–24, New York, NY, USA, 2003. ACM. ISBN 1-58113-588-2. doi: http://doi.acm.org/10.1145/781498.781501.

[13] M. Cintra and D. R. Llanos. Design space exploration of a software speculative parallelization scheme. *IEEE Trans. Parallel Distrib. Syst.*, 16(6):562–576, 2005. ISSN 1045-9219. doi: http://dx.doi.org/10.1109/TPDS.2005.69.

[14] W. Clinger. Twobit and Larceny benchmark suite. http://www.ccs.neu.edu/home/will/Twobit/.

[15] P. Costanza, C. Herzeel, and T. D'Hondt. Context-oriented Software Transactional Memory in Common Lisp. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 59–68, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-769-1. doi: http://doi.acm.org/10.1145/1640134.1640144.

[16] L. G. DeMichiel and R. P. Gabriel. The Common Lisp Object System: an overview. In *European conference on object-oriented programming on ECOOP '87*, pages 151–170, London, UK, 1987. Springer-Verlag. ISBN 0-387-18353-1.

[17] T. D'Hondt. Are Bytecodes an Atavism? In *Self-Sustaining Systems: First Workshop, S3 2008 Potsdam, Germany, May 15-16, 2008 Revised Selected Papers*, pages 140–155, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-89274-8. doi: http://dx.doi.org/10.1007/978-3-540-89275-5_8.

[18] J. Dongarra, I. Foster, G. Fox, W. Gropp, K. Kennedy, L. Torczon, and A. White, editors. *Sourcebook of parallel computing*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2003. ISBN 1-55860-871-0.

[19] D. P. Friedman and M. Wand. *Essentials of Programming Languages, 3rd Edition*. The MIT Press, 2008. ISBN 0262062798, 9780262062794.

[20] R. P. Gabriel. *Performance and evaluation of LISP systems*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1985. ISBN 0-262-07093-6.

[21] R. P. Gabriel and J. McCarthy. Queue-based multi-processing LISP. In *LFP '84: Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 25–44, New

York, NY, USA, 1984. ACM. ISBN 0-89791-142-3. doi: http://doi.acm.org/10.1145/800055.802019.

[22] R. P. Gabriel and G. L. Steele, Jr. A pattern of language evolution. In C. Herzeel, editor, *LISP50: Celebrating the 50th Anniversary of Lisp*, pages 1–10, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-383-9. doi: http://doi.acm.org/10.1145/1529966.1529967.

[23] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. pages 465–478, 2009. doi: http://doi.acm.org/10.1145/1542476.1542528.

[24] R. Goldman and R. P. Gabriel. Preliminary results with the initial implementation of Qlisp. In *LFP '88: Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 143–152, New York, NY, USA, 1988. ACM. ISBN 0-89791-273-X. doi: http://doi.acm.org/10.1145/62678.62696.

[25] R. Goldman and R. P. Gabriel. Qlisp: Parallel Processing in Lisp. *IEEE Software*, 6(4):51–59, 1989. doi: doi:10.1109/52.31652.

[26] Google. V8 JavaScript Engine. http://code.google.com/p/v8/.

[27] J. Gosling and H. McGilton. The Java Language Environment: A White Paper. Technical report, Sun Microsystems, Menlo Park, CA, USA, May 1996.

[28] R. H. Halstead. Multilisp: A language for concurrent symbolic computing. *ACM transactions on languages and systems*, 7 (4):501–538, 1985.

[29] T. Harris and K. Fraser. Language Support for Lightweight Transactions. *OOPSLA'03, Proceedings*, 2003. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/949343.949340.

[30] J. L. Hennessy and D. A. Patterson. *Computer Architecture: a quantitative approach*. Morgan Kaufmann Publishers, San Francisco, CA, USA, fourth edition, 2007.

[31] C. Herzeel, P. Costanza, and T. D'Hondt. Controlling dynamic parallelization through layered reflection. In "7th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing (POOSC'08)", 2008.

[32] C. Herzeel, P. Costanza, and T. D'Hondt. An Extensible Interpreter Framework for Software Transactional Memory. *Journal of Universal Computer Science*, 16(2):221–245, 2010.

[33] W. Horwat. Mozilla Javascript repository. http://mxr.mozilla.org/mozilla/source/js2/semantics/, 11 1999.

[34] P. Hudak. Yale Haskell '91 implementation. http://www.haskell.org/haskellwiki/Implementations, 10 1991.

[35] Intel. Intel Cilk SDK Programmer's Guide. http://software.intel.com/en-us/articles/intel-cilk/.

[36] A. C. Kay. The early history of Smalltalk. *SIGPLAN Not.*, 28(3):69–95, 1993. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/155360.155364.

[37] K. Kennedy and J. R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002. ISBN 1-55860-286-0.

[38] B. W. Kernighan and C. J. Van Wyk. Timing trials, or the trials of timing: experiments with scripting and user-interface languages. *Software: Practice and Experience*, 28(8):819–843, 1998. ISSN 0038-0644. doi: http://dx.doi.org/10.1002/(SICI)1097-024X(19980710)28:8⟨819::AID-SPE184⟩3.3.CO;2-F.

[39] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242, Berlin, Heidelberg, and New York, 1997. Springer-Verlag.

[40] T. Kistler and M. Franz. A Tree-Based Alternative to Java Byte-Codes. *Int. J. Parallel Program.*, 27(1):21–33, 1999. ISSN 0885-7458. doi: http://dx.doi.org/10.1023/A:1018740018601.

[41] T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the Java HotSpot™client compiler for Java 6. *ACM Trans. Archit. Code Optim.*, 5(1):1–32, 2008. ISSN 1544-3566. doi: http://doi.acm.org/10.1145/1369396.1370017.

[42] J. R. Larus and R. Rajwar. *Transactional Memory*. Morgan Claypool Publishers, USA, 2007. ISBN 1-59829-124-6.

[43] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *CC'03: Proceedings of the 12th international conference on Compiler construction*, pages 46–60, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-00904-3.

[44] J. McCarthy. History of LISP. In *History of programming languages I*, pages 173–185, New York, NY, USA, 1981. ACM. ISBN 0-12-745040-8. doi: http://doi.acm.org/10.1145/800025.1198360.

[45] H. G. Okuna and A. Gupta. Parallel Execution of OPSS in QLISP. Technical report, Stanford University, Stanford, CA, USA, 1987.

[46] R. B. Osborne. *Speculative Computation in Multilisp*. PhD thesis, Massachusetts Institute of Technology, December 1989.

[47] R. B. Osborne. Speculative Computation in Multilisp. In *LFP '90: Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 198–208, New York, NY, USA, 1990. ACM. ISBN 0-89791-368-X. doi: http://doi.acm.org/10.1145/91556.91644.

[48] M. Pall. The LuaJIT Project. http://luajit.org/.

[49] J. D. Pehoushek and J. S. Weening. Low-cost process creation and dynamic partitioning in Qlisp. In *Proceedings of the US/Japan workshop on Parallel Lisp on Parallel Lisp: languages and systems*, pages 182–199, New York, NY, USA, 1990. Springer-Verlag New York, Inc. ISBN 0-387-52782-6.

[50] S. L. Peyton Jones. Parallel implementations of functional programming languages. *Comput. J.*, 32(2):175–186, 1989. ISSN 0010-4620. doi: http://dx.doi.org/10.1093/comjnl/32.2.175.

[51] C. Queinnec. *Lisp in small pieces*. Cambridge University Press, New York, NY, USA, 1996. ISBN 0-521-56247-3.

[52] C. Quinones, C. Madriles, J. Sánchez, P. Marcuello, A. González, and D. M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. *SIGPLAN Not.*, 40(6):269–279, 2005. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1064978.1065043.

[53] R. Rugina and M. Rinard. Automatic parallelization of divide and conquer algorithms. In *PPoPP '99: Proceedings of the seventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 72–83, New York, NY, USA, 1999. ACM. ISBN 1-58113-100-3. doi: http://doi.acm.org/10.1145/301104.301111.

[54] P. Rundberg and P. Stenström. Low-cost thread-level data dependence speculation on multiprocessors. In *Proc. of the workshop on Multithreading Execution and Compilation at MICRO-33*, 2000.

[55] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2003. ISBN 0137903952.

[56] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Kelsey, W. Clinger, J. Reese, R. B. Findler, and J. Matthews. Revised^6 Report on the Algorithmic Language Scheme, September 2007.

[57] G. L. Steele, Jr. LAMBDA: The Ultimate Declarative. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1976.

[58] G. L. Steele, Jr. Debunking the "Expensive Procedure Call"" Myth or, Procedure Call Implementations Considered Harmful or, LAMDBA: The Ultimate GOTO. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1977.

[59] G. L. Steele, Jr. Rabbit: A Compiler for Scheme. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[60] G. L. Steele, Jr. Organizing Functional Code for Parallel Execution; or, foldl and foldr Considered Slightly Harmful. Keynote at the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP 2009), http://www.vimeo.com/6624203, 2009.

[61] G. L. Steele, Jr. and G. J. Sussman. Lambda: The Ultimate Imperative. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1976.

[62] G. L. Steele, Jr. and G. J. Sussman. The Art of the Interpreter or, The Modularity Complex (Parts Zero, One, and Two). Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1978.

[63] G. L. Steele, Jr and G. J. Sussman. Design of LISP-based processors, or SCHEME: A Dielectric LISP, or Finite Memories Considered Harmful, or LAMBDA: The Ultimate Opcode. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1979.

[64] G. J. Sussman and G. L. Steele, Jr. An Interpreter for Extended Lambda Calculus. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1975.

[65] J. Weening. *Parallel execution of Lisp programs*. PhD thesis, Stanford Comput. Sci. Rep. STANCS-89-1265, June 1989.