Partial Evaluation in Parallel

Charles Consel and Olivier Danvy
Research Report YALEU/DCS/RR-820
September 1990

This work is supported by the Darpa grant N00014-88-K-0573.

Partial Evaluation in Parallel (detailed abstract)

Charles Consel
Department of Computer Science
Yale University *
consel@cs.yale.edu

Olivier Danvy
Department of Computing and Info. Sciences
Kansas State University †
danvy@cis.ksu.edu

September 11, 1990

Abstract

Partially evaluating a procedural program amounts to building a series of mutually recursive specialized procedures. When a procedure call in the source program gets specialized into a residual call, the called procedure needs to be processed to occur in the residual program. Because the order of procedure definitions in the residual program is immaterial, it does not matter in which order these two events – building the residual call and building the residual procedure – are scheduled. Therefore, partial evaluation offers a very basic opportunity for an MIMD type of parallelism with shared global memory where in essence, the mutually recursive specialized procedures are built in parallel as specialization points are met and the relation binding source and residual procedures is globalized, to preserve its uniqueness.

We have translated a sequential partial evaluator written in T (a dialect of Scheme) into Mul-T (a parallel extension of T) by adding one semaphore for each specialization point and one future to construct the residual procedure in parallel with the current specialization. The resulting parallel partial evaluator has been observed to be faster than the sequential one in proportion to the size of the source program and to the number of specialized procedures in the residual program.

Our sequential partial evaluator is self-applicable. Because the semaphores and the future are run time operations, our parallel partial evaluator is still self-applicable. In principle it can be and in practice it has been used to generate parallel compilers, *i.e.*, specializers dedicated to an interpreter and processing its static and dynamic semantics in parallel, non trivially. Again, parallelism in dedicated specializers is determined by the size of the source program and the number of specialized procedures in the residual program.

Keywords

Specialization of programs, self-application, single-threading, Scheme, Mul-T, automatic generation of parallel programs, from interpreter to parallel compiler.

*P.O. Box 2158, New Haven, Connecticut 06520, USA.
†Manhattan, Kansas 66506, USA. This work was carried out during a visit to Yale in summer 1990.

1 Introduction

This paper describes how a program can be partially evaluated in parallel, based on a simple opportunity for parallelizing a sequential partial evaluator for sequential programs. The attractive properties of partial evaluation are still valid in a parallel setting, additionally to an increased efficiency. In particular, because our partial evaluator is self-applicable, we are able to derive a parallel compiler out of a sequential interpreter, automatically and in parallel. This amounts to refining the "syntax to dynamic semantics" mapping of [6] by making it operate in parallel.

This paper is organized as follows. Section 2 describes what partial evaluation is and how it is implemented. Section 3 points out a very natural opportunity to parallelize partial evaluation and why it is valid. However, parallelization requires to globalize and to side-effect the values that are single-threaded in the sequential partial evaluator. Section 3.1.2 investigates this aspect. Section 3.2 describes how the corresponding synchronization can be refined, thereby minimizing the needs to synchronize. Section 3.3 reports a series of benchmarks. Section 4 addresses how to generate a parallel compiler out of a sequential interpreter. This is achieved by self-application, *i.e.*, by partial evaluation of the partial evaluator with respect to an interpreter. Section 5 compares this approach with related works and puts this work into perspective.

2 Partial Evaluation

This section addresses the extensional and intensional aspects of partial evaluation. Partial evaluation specializes programs. It is achieved by executing these programs symbolically.

2.1 Partial evaluation: what

Partial evaluation is a program transformation technique that aims at specializing programs with respect to part of its input. By definition, running a residual program on the remaining input yields the same result as running the source program on the complete input.

$$\begin{aligned} run\ PE\ \langle program, incomplete\_input\rangle &= residue \\ run\ residue\ \langle remaining\_input\rangle &\equiv run\ program\ \langle complete\_input\rangle \end{aligned}$$

Partial evaluation is motivated by the frequent use of a program with constant patterns of input. Specializing this *source* program with respect to such a pattern yields a residual program that expects the remaining variable input to produce the result. A partial evaluator performs those parts of the source program that depend on the available input. The other computations are performed at run time.

Specialized programs can be expected to be more efficient than source programs since parts of the source program have been executed by the partial evaluator. These static (*i.e.*, partial evaluation time) computations occur only once, whereas the dynamic (*i.e.*, run time) computations can occur many times.

[1] offers a broad overview of partial evaluation, including a complete annotated bibliography of the field. This bibliography has been regularly updated ever since [17].

2.2 Partial evaluation: how

Partial evaluation is achieved by executing the program as much as the available input allow. The portions of the program that cannot be executed (because they depend on unavailable input) are reconstructed. This process yields a residual, specialized program.

We consider the partial evaluation of Scheme programs [14]. Our source and residual programs are written in Scheme. Our partial evaluator is written in Scheme as well and is self-applicable, *i.e.*, it can specialize itself non-trivially [4].

A Scheme program is a collection of mutually recursive procedures. According to some criteria which relate to making partial evaluation terminate [3, 2] but which fall outside the scope of this paper, some calls to these procedures are *specialization points* whereas all the other calls are *unfolded.* A specialization point potentially gives rise to many specialized versions of the source callee in the residual program.

Processing a call at a specialization point amounts to creating a residual call to a specialized procedure in the residual program and to constructing this specialized procedure. These two events are scheduled either breadth-first – the residual call is built right away and the procedure to be specialized is recorded for future treatment; or depth-first: the called procedure is processed right away, the specialized procedure is recorded for future reference, and then the residual call is built.

Due to this latitude in the traversal order, partial evaluation offers a very basic opportunity for parallelism.

3 Parallelism

It does not matter in which order residual calls are built and procedures are specialized because the order of procedure definitions in the residual program is immaterial. However parallelizing these constructions requires a synchronization, as adressed in section 3.1.1. This synchronization necessitates to globalize and to side-effect what were single-threaded values in the sequential partial evaluator, as addressed in section 3.1.2. Finally section 3.3 reports some measures of the parallel partial evaluator.

3.1 Parallelism: where

Section 2 ended on a basic opportunity for parallelism. Instead of complying with the breadth-first or depth-first strategy, the two actions – building the residual call and specializing the procedure – can be executed in parallel. Overall, partial evaluation is finished when the collection of residual procedures is completely built.

3.1.1 Synchronization

One synchronization is necessary, though, because two calls to the same specialization point with respect to the same static values should yield a residual call to the same residual procedure – otherwise the partial evaluator would build several identical residual procedures. At best it would give large and redundant residual programs; at worst (which occurs quite often, typically with a recursive residual procedure), it would not terminate.

Let us illustrate this last point and specialize the following source program

```
(define main    ; List(A) * List(A) * List(A) -> List(A) * List(A)
  (lambda (x y z)
    (cons (append x z) (append y z))))

(define append    ; List(A) * List(A) -> List(A)
  (lambda (x y)
    (if (null? x)
        y
        (cons (car x) (append (cdr x) y)))))
```

with respect to `z = (3 4 5)`. Because `z` is not the induction variable of procedure `append`, calls to `append` are classified as specialization points. Because this procedure is called three times with the same static argument during partial evaluation (two times in `main` and one time in `append`), the same residual procedure will be called three times in the residual program:

```
(define main-0    ; List(Num) * List(Num) -> List(Num) * List(Num)
  (lambda (x y)
    (cons (append-1 x) (append-1 y))))

(define append-1    ; List(Num) -> List(Num)
  (lambda (x)
    (if (null? x)
        '(3 4 5)
        (cons (car x) (append-1 (cdr x))))))
```

This specialized program expects the two remaining arguments to compute the same result as the source program, but somewhat faster, since `append-1` is monadic whereas `append` was dyadic. Without synchronization, partial evaluation would loop in the attempt to create infinitely many instances of `append-1`.

This example stresses the need for synchronizing accesses to the mapping between source program points, static values, and the corresponding residual procedures. For each source specialization point, a series of static values should determine a residual procedure uniquely. The following section describes how this uniqueness is preserved through parallelization.

3.1.2 Single-threaded values and the parallelization of functional programs

Schism is a partial evaluator written in pure Scheme, *i.e.*, it is side-effect free. It keeps a log recording which source procedures have been specialized with respect to which static values. This log is consulted and updated as the source program gets partially evaluated.

The only way to implement the log (consultation and updating) in a purely functional program is to pass it along to all the potential users and receive it updated from them, much like the store in the denotational specification of an imperative language. Since only one copy exists at any time during execution, the log is single-threaded [16]. In practice, single-threaded values are bound to global variables that are side-effected, to achieve efficiency while benefiting from the functional framework.

Single-threaded values assume a unique execution thread. Therefore, in general, they make it impossible to parallelize a program, since by definition, parallelizing amounts to creating several execution threads.

However, our log does not need to be updated sequentially. It only requires to have a unique representation at any time, as this representation gets updated. It does not matter where and when a

specialization point was first encountered with a particular set of static values. Simply, all the later encounters to this specialization point with the same set of static values should produce a residual call to the same specialized procedure. The first encounter triggers the specialization of a source procedure. All the other encounters refer to the specialized version of this procedure. Therefore, to parallelize Schism, we can globalize the log and synchronize its consultation and update using a semaphore. However, in contrast to the sequential case, side-effecting the log is not an optimization any more – it is a requirement.

This experiment illustrates how a functional program with single-threaded values can be parallelized when the order of access and update to these values does not matter. After globalization, the operations over the dummy value passed in place of the single-threaded one become semaphore operations that are essential to the correctness of the parallelized program.

In conclusion, sequential partial evaluation offers an opportunity for an MIMD type of parallelism with shared global memory.

3.2 Parallelism: how

Our partial evaluator – Schism – is written in the T dialect of Scheme [15]. T has been extended with Dijkstra's semaphores [9] and Halstead's futures [11] in the Mul-T system [13]. In spirit, this extension encourages one to spread a few parallel constructs in an existing sequential program to get a parallel one. Let us describe how to parallelize Schism.

Parallelizing the breadth-first or depth-first traversal of the source program is achieved using a future construct. Instead of sequentializing the construct of the residual call and the specialization of the called procedure, we construct the residual procedure with a promise to specialize the called procedure, and we construct the residual call right away.

Synchronizing accesses and updates to the log is achieved with a semaphore. This makes it possible to preserve the unicity of the relationship between source procedures, static values, and residual procedures that was ensured, in the sequential case, with a single-threaded log.

However, this synchronization is too coarse, and as such it strangles the whole partial evaluation process, since there is no need to synchronize when two distinct specialization points are reached. Synchronization only matters when the same specialization point is reached concurrently. Therefore, we refine the synchronization by associating one semaphore to each source specialization point.

To summarize, we parallelize Schism by wrapping a future around the construction of a residual procedure and by associating one semaphore to each specialization point.

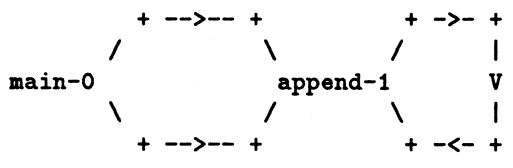

Figure 1: Task graph. (to be drawn better with xfig)

One can draw the dependency graph of the parallel processes during partial evaluation, making a node stand for each task constructing a residual procedure and one vertex stand for each access to the log. Figure 1 displays the task graph corresponding to the example above. Processing `main-0` spawns the processing of `append-1` that would spawn the processing of `append-1` if this was not done already.

As can be noticed, this graph is identical to the call graph of the residual program (before trivial post-unfolding [2]).

3.3 Benchmarks for parallel partial evaluation

This section presents the results of specializing a program in parallel.

$$run\ PE_p\ <program, incomplete\_input>\ =\ residue$$

PE_p is the parallel partial evaluator. It has been run on an interpreter for the applied lambda-calculus and a pattern matcher in lists [8]. We are currently extending these measures to pattern matching in strings [5] and the interpretation of Algol and of Prolog programs [6, 7].

Table 1 displays the size (in characters) of the source programs. This component takes a part in the overall performance.

	Size (Chars)
Lambda Interpreter	4865
Matcher	3906
Specializer	57626

Table 1: Size of source programs

The run time figures are given in seconds with two decimals. They were obtained on the Encore Multimax machine with the version 4.0 of Mul-T [13]. They exclude the time used for garbage collection and have the usual uncertainty connected to CPU measures.

Tables 2 and 3 display the performance statistics for the specialization of the Lambda interpreter and the pattern matcher, respectively. In both cases, specialization does not improve when the number of processors is increased. This is due to the poor latent parallelism of these applications: specializing the Lambda interpreter only yields two residual procedures, and only one for the pattern matcher. Two futures have been created for the former; only one has been created for the latter (see second line of tables 2 and 3).

Number of Processors	1	2	4	8
Run time	6.94	6.32	6.19	6.10
Number of Futures	2	2	2	2
Idle cycles	0	32671	99108	229593

Table 2: Specialization of the Lambda interpreter

Number of Processors	1	2	4	8
Run time	5.22	5.01	5.09	5.01
Number of Futures	1	1	1	1
Idle cycles	0	27177	82762	189866

Table 3: Specialization of the pattern matcher

These tables confirm our initial intuition that parallel partial evaluation performs well for big programs with many specialization points. When there are too few opportunities for parallelism, speedup is not linear (with respect to the number of processors) because some processsors are idle.

4 Self-Application

Self-applying a partial evaluator aims at optimizing the process of specialization. In a nutshell, repeated partial evaluation of a program with respect to different values can be optimized by first generating a specializer dedicated to this program and second by running this dedicated specializer repeatedly. This is captured in the following equations.

A partial evaluator specializes programs:

$$run\ PE\ \langle program, data\rangle \;=\; residue$$

Partial evaluation can be optimized by preliminary generation of a dedicated specializer:

$$\begin{aligned} run\ PE\ \langle PE,\ \langle program, \_\rangle\rangle &= DS \\ run\ DS\ \langle data\rangle &= residue \end{aligned}$$

Partial evaluation can be further optimized by preliminary generation of a generator of dedicated specializers:

$$\begin{aligned} run\ PE\ \langle PE,\ \langle PE, \_\rangle\rangle &= Curry \\ run\ Curry\ \langle program, \_\rangle &= DS \end{aligned}$$

Curry takes its name from the fact that it curries a program intensionally. In other words, it may take a two-argument program and return a new program expecting the first argument and yielding another program that expects the second argument and produces the result of the original program if it is applied to both arguments. This transformation is intensional because it operates on the representation of these programs.

Self-application has received a great deal of attention due to the following particular case. If the source program is an interpreter, the dedicated specializer has the functionality of a compiler [10]. Correspondingly, if the source program is the partial evaluator itself, applying the dedicated specializer to an interpreter makes it have the functionality of a compiler generator.

Our sequential partial evaluator is self-applicable. Now that we have a parallel partial evaluator, how can we self-apply it and what does self-application yield? The following sections answer these two questions.

4.1 Self-application in parallel

Self-application is only the particular case where the source program is the partial evaluator itself. Therefore, the sequential partial evaluator can be specialized in parallel, yielding the same sequential dedicated specializer, modulo the order of procedure definitions:

$$run\ PE_p\ <PE,\ <program, \_>>\ \equiv\ run\ PE\ <PE,\ <program, \_>>$$

The parallel apparatus described in section 3.2 aims at making promises about the definition of residual procedures and at synchronizing the access to the global log. Both these operations depend on the static values a program is specialized with respect to. Because these values are not available at self-application time, these operations need to be classified as dynamic, *i.e.*, they will be performed only when the dedicated specializer runs.

Therefore, by declaring the semaphore operations and the future construction to be dynamic algebraic operators, we can specialize the parallel partial evaluator, either sequentially or in parallel:

$$run\ PE\ <PE_p,\ <program, \_>>\ \equiv\ run\ PE_p\ <PE_p,\ <program, \_>>$$

This illustrates how, in a sense, a partial evaluator only needs to know about the static subset of its input language.

4.2 Parallel dedicated specializers

Because the operations managing parallelism are deferred until the dedicated specializer runs, self-application produces residual programs with parallelization points. In other words, dedicated specializers run in parallel as well.

To summarize, parallel partial evaluation can be optimized by preliminary parallel generation of a dedicated, parallel specializer:

$$\begin{aligned} run\ PE_p\ <PE_p,\ <program, \_>> &= DS_p \\ run\ DS_p\ <data> &= residue \end{aligned}$$

Parallel partial evaluation can be further optimized by preliminary generation of a parallel generator of dedicated, parallel specializers:

$$\begin{aligned} run\ PE_p\ <PE_p,\ <PE_p, \_>> &= Curry_p \\ run\ Curry_p\ <program, \_> &= DS_p \end{aligned}$$

In the particular case of interpreters, we obtain parallel compilers that are generated in parallel. Let us compare them and their generation with their sequential counterparts.

4.3 Benchmarks for parallel self-application

The following measures have been taken under the same conditions as in section 3.3. They account for the results of generating a specializer dedicated to a program.

$$run\ PE_p\ <PE_p,\ <program, \_>> \ = \ DS_p$$

Tables 2 and 3 display the performance statistics for self-application with respect to the Lambda interpreter and the pattern matcher, respectively. As can be observed, these applications improve when more processors are provided. With eight processors these applications are two to three times faster than with one processor.

Number of Processors	1	2	4	8
Run time	267.00	142.51	93.25	81.08
Number of Futures	21	21	21	21
Idle cycles	0	105092	570621	2049373

Table 4: Self-application with respect to the Lambda interpreter

Number of Processors	1	2	4	8
Run time	416.02	223.64	165.84	161.45
Number of Futures	24	24	24	24
Idle cycles	0	142327	1292846	4664247

Table 5: Self-application with respect to the pattern matcher

Let us also state the following observation. The more processors are provided, the less run times improve. This is confirmed by the idle cycles (third line of the tables) whose numbers increase rapidly with the number of processors. Likely, more parallelism could be introduced even when specializing large programs like the specializer itself.

On the other hand, these measures stress the speedup obtained using self-application. The dedicated specializers run faster than the corresponding direct partial evaluation. The likelihood of this improvement originally motivated self-application [10, 12]. However, from the parallelism viewpoint, self-application does not change anything: the parallelism shown in specializing the pattern matcher or the Lambda interpreter is the same as the one of the dedicated specializers.

5 Conclusion and Issues

We have identified a very simple opportunity for parallelism in a partial evaluator for procedural programs. We have implemented it by extending an existing sequential partial evaluator, and we have measured its effect on typical specializations. We have pointed out how to make this parallelization compatible with self-application and we have generated parallel specializers dedicated to sequential programs, which we believe is an original way to generate parallel programs out of sequential ones.