A Study on the Practicality of Poly-Controlled Partial Evaluation

Claudio Ochoa and Germán Puebla

School of Computer Science
Technical University of Madrid
Madrid, Spain
{*claudio,german*}*@fi.upm.es*

Abstract

Poly-controlled partial evaluation (PCPE) is a flexible approach for specializing logic programs, which has been recently proposed. It takes into account *repertoires* of global control and local control rules instead of a single, predetermined, combination. Thus, *different* global and local control rules can be assigned to different call patterns, obtaining results that are *hybrid* in the sense that they cannot be obtained using a single combination of control rules, as traditional partial evaluation does. PCPE can be implemented as a *search-based* algorithm, producing *sets* of candidate specialized programs (many of them hybrid), instead of a single one. The quality of each of these programs is assessed through the use of different fitness functions, which can be *resource aware*, taking into account multiple factors such as run-time, memory consumption, and code size of the specialized programs, among others. Although PCPE is an appealing approach, it suffers from an inherent blowup of its search space when implemented as a search-based algorithm. Thus, in order to be used in practice, and to deal with realistic programs, we must be able to prune its search space without losing the interesting solutions. The contribution of this work is two-fold. On one hand we perform an experimental study on the heterogeneity of solutions obtained by search-based PCPE, showing that the solutions provided behave very differently when compared using a fitness function. Note that this is important since otherwise the cost of producing a large number of candidate specializations would not be justified. The second contribution of this work is the introduction of a technique for pruning the search space of this approach. The proposed technique is easy to apply and produces a considerable reduction of the size of the search space, allowing PCPE to deal with a reasonable number of benchmark programs. Although pruning is done in a *heuristic* way, our experimental results suggest that our heuristic behaves well in practice, since the fitness value of the solutions obtained using pruning coincide with the fitness value of the solution obtained when no pruning is applied.

Keywords: Partial Evaluation, Control Strategies, Resource Awareness, Program Optimization, Pruning Techniques

1 Introduction

The aim of partial evaluation (*PE*) is to specialize a program w.r.t. part of its input, which is known as the *static data*[11]. The quality of the code generated by partial evaluation greatly depends on the *control strategy* used. Unfortunately, the existence of sophisticated control rules which behave (almost) optimally for all programs is still far from reality. Poly-controlled partial evaluation [15] (*PCPE*) attempts to cope with this problem by employing a *set* of global and local control rules instead of a predetermined combination (as done in traditional partial evaluation algorithms). This allows using *different* global and local control rules for different call patterns (atoms). Thus, PCPE can produce specialized programs that are *not achievable* by traditional partial evaluation using any of the considered local and global control rules in isolation.

In [15], two algorithms for implementing PCPE were introduced. One of them uses a function called *pick* to decide *a priori* which (global and local) control strategies are to be applied to every atom. The second one applies a number of pre-selected control rules to every atom, generating several candidate specializations, and decides *a posteriori* which specialization is the best one by empirically comparing the final configurations (candidate specializations) using a fitness function, possibly taking into account factors such as size of the specialized program and time- and memory-efficiency of such a specialized program. Since choosing a good *Pick* function can be a very hard task, and in the need of a proof of concept of the idea of PCPE, we have implemented the second algorithm (leaving the first one for future work), although this algorithm is less efficient in terms of size of the search space.

Among the main advantages of PCPE we can mention:

It can obtain better solutions than traditional PE: In [15], preliminary experiments showed that PCPE produced *hybrid* solutions with better fitness value than any of the solutions achievable by traditional PE, for a number of different resource-aware fitness functions. *Hybrid* solutions are not achievable by traditional partial evaluation, since *different* global and local control rules are applied to *different* call patterns.

It is a resource-aware approach: in traditional PE, existing control rules focus on time-efficiency by trying to reduce the number of *resolution steps* which are performed in the residual program. Other factors such as the size of the compiled specialized program, and the memory required to run the residual program are most often neglected—some relevant exceptions being the works in [4],[3]—. In addition to potentially generating larger programs, it is well known that partial evaluation can slow-down programs due to lower level issues such as clause indexing, cache sizes, etc. PCPE, on the other hand, makes use of *resource aware* fitness functions to choose the best solution from a set of candidate solutions.

It is more user-friendly: existing partial evaluators usually provide several global and local control strategies, as well as many other parameters (global trees, computation rules, etc.) directly affecting the quality of the obtained solution. For a novice user, it is extremely hard to find the right combination of parameters in order to achieve the desired results (reduction of size of compiled code, reduction of execution time, etc.). Even for an experienced user, it is rather difficult to predict the behavior of partial evaluation, especially in terms of space-efficiency (size of the residual program). PCPE allows the user to simultaneously experiment with different combinations of parameters in order to achieve a specialized program with the desired characteristics.

It performs online partial evaluation: as opposed to other approaches (e.g. [3]), PCPE performs *online* partial evaluation, and thus it can take advantage of the great body of work available for *online* partial evaluation of logic programs.

Unfortunately, PCPE is not the panacea, and it has a number of disadvantages. The main drawback of this approach is that, *when implemented as a search-based* *algorithm*, its search space suffers from an inherent exponential blowup since given a configuration, the number of successors can be as high as the number of combinations of local and global control rules considered. As a direct consequence, the specialization time of PCPE is higher than its PE counterpart.

After getting acquainted for the first time with the basic idea of poly-controlled partial evaluation, probably two questions come up immediately to our mind:

(i) does PCPE provides a wide range of solutions? I.e., is the set of obtained solutions heterogeneous enough to offer us a wide set of candidate solutions to choose from?

(ii) is PCPE feasible in practice? I.e., since there is an exponential blowup of the search space, is it possible to perform some pruning in order to deal with realistic programs without losing the interesting solutions?

Throughout this work we address these two questions, providing some experimental results to help us justify our allegations.

2 Background

We assume some basic knowledge on the terminology of logic programming. See for example [12] for details.

Very briefly, an *atom* A is a syntactic construction of the form $p(t_1, \ldots, t_n)$, where p/n, with $n \geq 0$, is a predicate symbol and $t_1, \ldots, t_n$ are terms. The function *pred* applied to atom A, i.e., $pred(A)$, returns the predicate symbol p/n for A. A *clause* is of the form $H \leftarrow B$ where its head H is an atom and its body B is a conjunction of atoms. A *definite program* is a finite set of clauses. A *goal* (or query) is a conjunction of atoms.

Two terms t and t' are *variants*, denoted $t \approx t'$, if there exists a renaming ρ such that $t\rho = t'$. We denote by $\{X_1 \mapsto t_1, \ldots, X_n \mapsto t_n\}$ the *substitution* σ with $\sigma(X_i) = t_i$ for all $i = 1, \ldots, n$ (with $X_i \neq X_j$ if $i \neq j$) and $\sigma(X) = X$ for any other variable X, where t_i are terms. A *unifier* for a finite set S of simple expressions is a substitution θ if $S\theta$ is a singleton. A unifier θ is called *most general unifier* (*mgu*) for S, if for each unifier σ of S, there exists a substitution γ such that $\sigma = \theta\gamma$.

2.1 Basics of Partial Evaluation in LP

Partial evaluation of LP is traditionally presented in terms of SLD semantics. We briefly recall the terminology here. The concept of *computation rule* is used to select an atom within a goal for its evaluation.

Definition 2.1 A *computation rule* is a function $\mathcal{R}$ from goals to atoms. Let G be a goal of the form $\leftarrow A_1, \ldots, A_R, \ldots, A_k$, $k \geq 1$. If $\mathcal{R}(G) = A_R$ we say that A_R is the *selected* atom in G.

The operational semantics of definite programs is based on derivations [12].

Definition 2.2 [derivation step] Let G be $\leftarrow A_1, \ldots, A_R, \ldots, A_k$. Let $\mathcal{R}$ be a computation rule and let $\mathcal{R}(G) = A_R$. Let $C = H \leftarrow B_1, \ldots, B_m$ be a renamed apart clause in P. Then G' is *derived* from G and C via $\mathcal{R}$ if the following conditions hold:

$$\theta = mgu(A_R, H)$$
$$G' \text{ is the goal } \leftarrow \theta(A_1, \ldots, A_{R-1}, B_1, \ldots, B_m, A_{R+1}, \ldots, A_k)$$

As customary, given a program P and a goal G, an *SLD derivation* for $P \cup \{G\}$ consists of a possibly infinite sequence $G = G_0, G_1, G_2, \ldots$ of goals, a sequence $C_1, C_2, \ldots$ of properly renamed apart clauses of P, and a sequence $\theta_1, \theta_2, \ldots$ of mgus such that each G_{i+1} is derived from G_i and C_{i+1} using θ_{i+1}.

A derivation step can be non-deterministic when A_R unifies with several clauses in P, giving rise to several possible SLD derivations for a given goal. Such SLD derivations can be organized in *SLD trees*. A finite derivation $G = G_0, G_1, G_2, \ldots, G_n$ is called *successful* if G_n is empty. In that case $\theta = \theta_1\theta_2 \ldots \theta_n$ is called the computed answer for goal G. Such a derivation is called *failed* if it is not possible to perform a derivation step with G_n.

In partial evaluation, SLD semantics is extended in order to also allow *incomplete derivations* which are finite derivations of the form $G = G_0, G_1, G_2, \ldots, G_n$ and where no atom is selected in G_n for further resolution. This is needed in order to avoid (local) non-termination of the specialization process. Also, the substitution $\theta = \theta_1\theta_2 \ldots \theta_n$ is called the computed answer substitution for goal G. An *incomplete SLD tree* possibly contains incomplete derivations.

In order to compute a *partial evaluation* (PE) [11], given an input program and a set of atoms (goals), the first step consists in applying an *unfolding rule* to compute finite incomplete SLD trees for these atoms. Then, a set of *resultant*s or residual rules are systematically extracted from the SLD trees.

Definition 2.3 [unfolding rule] Given an atom A, an unfolding rule computes a set of finite SLD derivations $D_1, \ldots, D_n$ (i.e., a possibly incomplete SLD tree) of the form $D_i = A, \ldots, G_i$ with computer answer substitution θ_i for $i = 1, \ldots, n$ whose associated *resultants* are $\theta_i(A) \leftarrow G_i$.

Therefore, this step returns the set of resultants, i.e., a program, associated to the root-to-leaf derivations of these trees. The set of resultants for the computed SLD tree is called a *partial evaluation* for the initial goal (query). The partial evaluation for a set of goals is defined as the union of the partial evaluations for each goal in the set. We refer to [8] for details.

In order to ensure the local termination of the PE algorithm while producing useful specializations, the unfolding rule must incorporate some non-trivial mechanism to stop the construction of SLD trees. Nowadays, well-founded orderings (wfo) [2,13] and well-quasi orderings (wqo) [16,9] are broadly used in the context of on-line partial evaluation techniques (see, e.g., [6,10,16]).

In addition to local termination, an *abstraction operator* is applied to properly add the atoms in the right-hand sides of resultants to the set of atoms to be partially evaluated. This abstraction operator performs the *global control* and is in charge of guaranteeing that the number of atoms which are generated remains finite. This is done by replacing atoms by more general ones, i.e., by losing precision in order to guarantee termination. The abstraction phase yields a new set of atoms, some of which may in turn need further evaluation and, thus, the process is iteratively repeated while new atoms are introduced.

3 Poly-Controlled Partial Evaluation

Traditional algorithms for partial evaluation (PE) of logic programs (LP) are parametric w.r.t. the *global control* and *local control* rules[1] . In these algorithms, once a specialization strategy has been selected, it is applied to all call patterns in the residual program. However, it is well known that several control strategies exist which can be of interest in different circumstances. It is indeed a rather difficult endeavor to find a specialization strategy which behaves well in all settings. Thus, rather than considering a single specialization strategy, at least in principle one can be interested in applying *different* specialization strategies to *different* atoms (call patterns). Unfortunately, this is something which existing algorithms for PE do not cater for. Poly-controlled partial evaluation (PCPE) [15] fills this gap by allowing the use of *a set* of specialization strategies instead of a predetermined one.

3.1 A Search-Based *Poly-Controlled Partial Evaluation Algorithm*

Algorithm 1 shows a *search-based poly-controlled* partial evaluation algorithm. In this algorithm, a configuration $Conf_i$ is a pair $\langle S_i, H_i\rangle$ s.t. S_i is the set of atoms yet to be handled by the algorithm and H_i is the set of atoms already handled by the algorithm. Indeed, in H_i not only we store atoms A_i but also the result A'_i of applying global control to such atoms and the unfolding rule $Unfold$ which has been used to unfold A_i, i.e., members of H_i are tuples of the form $\langle A_i, A'_i, Unfold\rangle$. We store $Unfold$ in order to use exactly such unfolding rule during the code generation phase. Correctness of the algorithm requires that each A'_i is an *abstraction* of A_i, i.e., $A_i = A'_i\theta$. Algorithm 1 employs two auxiliary data structures. One is *Confs*, which contains the configurations which are currently being explored. The other one is *Sols*, which stores the set of solutions currently found by the algorithm. As it is well known, the use of different data structures for *Confs* provides different traversals of the search space. In our implementation of this algorithm in `CiaoPP` [7], we have used both a stack and a queue, traversing the search space in a depth-first and a breadth-first fashion, respectively.

Given a set of atoms S which describe the potential queries to the program, the initial configuration is of the form $\langle S, \emptyset\rangle$. In each iteration of the algorithm, a configuration $\langle S_i, H_i\rangle$ is popped from *Confs* (line 6), and an atom A_i from S_i is selected (line 7). Then, several combinations of global control ($Abstract \in \mathcal{G}$) and local control ($Unfold \in \mathcal{U}$) rules, respectively, are applied (lines 11 and 12). Each application builds an SLD-tree for A'_i, a generalization of A_i as determined by *Abstract*, using the corresponding unfolding rule *Unfold*. Once the SLD-tree τ_i is computed, the leaves in its resultants, i.e., the atoms in the residual code for A'_i are collected by the function *leaves* (line 14). Those atoms in *leaves*(τ_i) which are not a variant of an atom handled in previous iterations of the algorithm are added to the set of atoms to be considered (S_{i+1}) and pushed on *Confs*. We use

[1] From now on, we call any combination of global and local control rules a *specialization strategy.*

Algorithm 1 Search-Based Poly-Controlled Partial Evaluation Algorithm

Input: Program P
Input: Set of atoms of interest S
Input: Set of unfolding rules $\mathcal{U}$
Input: Set of generalization functions $\mathcal{G}$
Output: Set of partial evaluations $Sols$

```
1:  $H_0 = \emptyset$
2:  $S_0 = S$
3:  create($Confs$); $Confs$ = push($\langle S_0, H_0 \rangle$, $Confs$)
4:  $Sols = \emptyset$
5:  repeat
6:     $\langle S_i, H_i \rangle$ = pop($Confs$)
7:     $A_i = Select(S_i)$
8:     $Candidates = \{\langle Abstract, Unfold \rangle \mid Abstract \in \mathcal{G}, Unfold \in \mathcal{U}\}$
9:     repeat
10:       $Candidates = Candidates - \{\langle Abstract, Unfold \rangle\}$
11:       $A'_i = Abstract(H_i, A_i)$
12:       $\tau_i = Unfold(P, A'_i)$
13:       $H_{i+1} = H_i \cup \{\langle A_i, A'_i, Unfold \rangle\}$
14:       $S_{i+1} = (S_i - \{A_i\}) \cup \{A \in leaves(\tau_i) \mid \forall\ \langle B, \_, \_ \rangle \in H_{i+1}\ .\ B \not\equiv A\}$
15:       if $S_{i+1} = \emptyset$ then
16:          $Sols = Sols \cup \{H_{i+1}\}$
17:       else
18:          push($\langle S_{i+1}, H_{i+1} \rangle$,$Confs$)
19:       end if
20:    until $Candidates = \emptyset$
21:    $i = i + 1$
22: until empty_stack($Confs$)
```

$B \equiv A$ to denote that B and A are *variants*, i.e., they are equal modulo variable renaming. The process terminates when the stack of configurations to handle is empty, i.e. all final configurations have been reached. The specialized program corresponds to $\bigcup_{\langle A, A', Unfold \rangle \in H_n} resultants(A', Unfold)$, where the function *resultants* is parametric w.r.t. the unfolding rule.

Note that in this algorithm, once an atom A_i is abstracted into A'_i, code for A'_i will be generated, and it will not be abstracted any further no matter which other atoms are handled in later iterations of the algorithm. As a result, the set of atoms for which code is generated are not guaranteed to be *independent*. Two atoms are independent when they have no common instance. However, the pairs in H uniquely determine the version used at each program point. Since code generation produces a new predicate name per entry in H, independence is guaranteed, and thus the specialized program will not produce more solutions than the original one.

As mentioned in [15], one could think of a similar algorithm deciding *a priori* a control strategy to be applied to each atom. This algorithm would be more similar to the traditional PE algorithm, employing possibly different control rules for differ-

```
:- module(_,[rev/2],[]).
:- entry rev([_,_|L],R).

rev([],[]).
rev([H|L],R) :-
    rev(L,Tmp),
    app(Tmp,[H],R).

app([],L,L).
app([X|Xs],Y,[X|Zs]) :-
    app(Xs,Y,Zs).
```

(a)

Input query	#solutions
rev(L,R)	6
rev([_|L],R)	48
rev([_,_|L],R)	117
rev([_,_,_|L],R)	186
rev([_,_,_,_|L],R)	255
rev([1|L],R)	129
rev([1,2|L],R)	480

(b)

Fig. 1. The `nrev` example and the number of solution generated by PCPE

ent atoms. Unfortunately, it is not clear how this decision can be made, so instead Algorithm 1 generates several candidate partial evaluations and then decides *a posteriori* which specialized program to use. Clearly, generating all possible candidate specialized programs is more costly than computing just one. However, selecting the best candidate a posteriori allows to make much more informed decisions than selecting it a priori.

3.2 Exponential Blowup of the Search Space

Given that Algorithm 1 allows different combinations of specialization strategies, given a configuration, there are several successor configurations. This can be interpreted as, given $\mathcal{G}=\{A_1,\ldots,A_j\}$ and $\mathcal{U}=\{U_1,\ldots,U_i\}$, there is a *set* of transformation operators $T^{A_1}_{U_1},\ldots,T^{A_1}_{U_i},\ \ldots,T^{A_j}_{U_i}$. Thus, in the *worst* case, given a set of unfolding rules $\mathcal{U}=\{\mathit{Unfold}_1,\ldots,\ \mathit{Unfold}_i\}$, and a set of abstraction functions $\mathcal{G}=\{\mathit{Abstract}_1,\ldots,\ \mathit{Abstract}_j\}$, there are $i\times j$ possible combinations. As already mentioned, this represents an inherent exponential blowup in the size of the search space, and it makes the algorithm impractical for dealing with realistic programs.

Of course, several optimizations can be done to the base algorithm shown above, in order to deal with this problem. A first obvious optimization is to eliminate equivalent configurations which are descendants of the same node in the search tree. I.e., it is often the case that given a configuration *Conf* there are more than one T^A_U and $T^{A'}_{U'}$ with $(A,U)\neq(A',U')$ s.t. $T^A_U(\mathit{Conf})=T^{A'}_{U'}(\mathit{Conf})$. This optimization is easy to implement, not very costly to execute, and reduces search space significantly.

However, even with this optimization, a simple experiment shows the magnitude of this problem. Let us consider the program in Listing 1(a), which implements a naive reverse algorithm.

In this experiment, let us choose the set of global control rules $\mathcal{G}=\{\mathit{dynamic}$, $\mathit{hom\_emb}\}$. The *hom_emb* global control rule is based on homeomorphic embedding [8,9] and flags atoms as potentially dangerous (and are thus generalized) when they homeomorphically embed any of the previously visited atoms at the global control level. Then, *dynamic* is the most abstract possible global control rule, which abstracts away the value of all arguments of the atom and replaces them with distinct variables. Also, let us choose the set of local control rules $\mathcal{U}=\{\mathit{one\_step}$, $\mathit{df\_hom\_emb\_as}\}$. The rule *one_step* is the simplest possible unfolding rule which always performs just one unfolding step for any atom. Finally, *df_hom_emb_as* is an unfolding rule based on homeomorphic embedding. More details on this unfolding rule can be found in [14]. It can handle external predicates safely and can perform non-leftmost unfolding as long as unfolding is safe (see [1]) and *local* (see [14]).

In `CiaoPP` [7], the description of initial queries (i.e., the set of atoms of interest S in Algorithm 1) is obtained by taking into account the set of predicates exported by the module, in this case `rev/2`, possibly qualified by means of *entry* declarations. For example, the *entry* declaration in Listing 1(a) is used to specialize the naive reverse procedure for lists containing *at least* two elements.

Table (b) of Figure 1 shows the number of candidate solutions generated by Algorithm 1 (eliminating equivalent configurations in the search tree), for several *entry* declarations. As can be observed in the table, as the length of the list provided as entry grows, the number of candidate solutions computed quickly grows. Furthermore, if the elements of the input list are static, then the number of candidates grows even faster, as can be seen in the last two rows in Table 1, where we provide the first elements of the list. From this small example, it is clear that, in order to be able to cope with realistic Prolog programs, it is mandatory to reduce the search space. In Section 5 we propose a technique to do so.

4 Heterogeneity of PCPE *Hybrid* Solutions

As mentioned before, Algorithm 1 produces a set of candidate solutions. Of these, a few of them are *pure*, in the sense that they can be obtained via traditional PE (i.e., they apply the same control strategy to all atoms in the residual program), and the rest are *hybrid*, in the sense that they apply different specialization strategies to different atoms. In this section, we try to determine how heterogeneous are the fitness values of the different solutions obtained by PCPE.

4.1 Choosing Adequate Sets of Global and Local Control Rules

The question of whether the solutions obtained by PCPE are heterogeneous w.r.t. their fitness values depends, in a great deal, on the particular choice of specialization strategies to be used, as well as on the arity of the sets $\mathcal{G}$ and $\mathcal{U}$ of control rules. We can expect that by choosing control rules different enough, the candidate solutions will be also very different, and viceversa. To see this, think for a moment that we choose $\mathcal{U} = \{det, lookahead\}$ where both *det* and *lookahead* are purely determinate [6,5]—i.e., they select atoms matching a single clause head—, the difference being that *lookahead* uses a ”look-ahead” of a finite number of computation steps to detect further cases of determinacy [6]. Given that both rules are based on determinate unfolding, and this is considered a very conservative technique, it is highly probable that this particular choice of local control rules will not contribute to finding heterogeneous solutions. A better idea will be then to choose one unfolding rule that is conservative, and another one that is aggressive. An example of an aggressive local control rule would be one performing *non-leftmost* unfolding. The same reasoning can be done when selecting the global control rules, we could select one rule that is very precise—while guaranteeing termination—, and a very imprecise global control rule.

Benchmark	Input Query	speedup				
		Vers	Fitness	Mean	St Dev	Diam
example_pcpe	main(_,_,2,_)	27	1.56	0.87	0.21	0.99
permute	permute([1,2,3,4,5,6],L)	70	1.31	1.15	0.48	1.16
nrev	rev([_,_,_,_|L],R)	255	1.09	0.66	0.15	0.51
advisor	what_to_do_today(_,_,_)	14	1.68	1.31	0.67	0.97
relative	relative(john,X)	61	18.01	3.45	4.84	16.37
ssuply	ssupply(_,_,_)	31	5.15	1.84	1.82	4.72
transpose	transpose([[_,_,_,_,_,_,_,_,_],_,_],_)	154	2.62	0.87	0.30	2.13
overall		**87.4**	**4.49**	**1.45**	**1.21**	**3.83**

Table 1
PCPE statistics over different benchmarks (speedup)

4.2 Heterogeneity of the Fitness of PCPE Solutions

Once we select an appropriate set of control rules for PCPE, we need to determine whether the fitness of the solutions we obtain are heterogeneous. With this purpose, we have ran some experiments over a set of benchmarks and different fitness functions, in order to collect statistical facts such as *Standard Deviation* and *Diameter* that can help us to determine how different are the obtained solutions. In our experiments, as mentioned in Section 3, we have used a set of global control rules $\mathcal{G}$={*dynamic*, *hom_emb*} and a set of local control rules $\mathcal{U}$={*one_step*, *df_hom_emb_as*}. Besides, we used different fitness functions already introduced in [15]. For reasons of space, we will show some of the results obtained when using the following fitness functions:

speedup compares programs based on their time-efficiency, measuring run-time speedup w.r.t. the original program. When using this fitness function, the user needs to provide a set of run-time queries with which to time the execution of the program[2]. This fitness function is computed as

$$speedup = T_{orig}/T_{spec},$$

where T_{spec} is the execution time taken by the specialized program to run the given run-time queries, and T_{orig} the time taken by the original program.

reduction compares programs based on their space-efficiency, measuring reduction of size of compiled bytecode w.r.t. the original program. It is computed as

$$reduction = (S_{orig} - S_{empty}) / \ (S_{spec} - S_{empty}),$$

where S_{spec} is the size of the compiled bytecode of the specialized program, S_{orig} is the size of the compiled bytecode of the original program, and S_{empty} is the size of the compiled bytecode of an empty program.

In Table 1 we can observe, for a number of benchmarks, the collected statistics when using *speedup* [15] as a fitness function. As mentioned before, the number of versions obtained is tightly related to several factors, such as the number and kind of control rules used, as well as the initial input queries used to specialize each program. For this particular experiment, PCPE generated a mean of 87 candidate solutions per benchmark. In most cases we can observe that both the fitness of the

[2] Though the issue of finding representative run-time queries is an interesting research topic in its own right, it is out of the scope of this paper to automate such process.

best solution and the mean fitness are over 1, meaning that a speedup is achieved when comparing the obtained solutions w.r.t. the original program. In some cases, the mean speedup is below 1, indicating that many of the solutions are bad and get a slowdown w.r.t. the original program. Let us take *transpose*, for example. In this particular benchmark, we can see that most of the 154 final solutions are slower than the original program, meaning that it is easy to specialize this program with different control strategies and obtain a solution that runs slower than the original program. Note however, that the best solution obtained by PCPE is 2.62 faster than the original program.

In order to answer our initial question, i.e., whether does PCPE provide a wide range of solutions, the columns we are interested in looking at are *St Dev* and *Diameter*. *St Dev* stands for standard deviation, and measures how spread out the values in a data set are. *Diameter* measures the difference of fitness among (any of) the best solution(s) when compared to (any of) the worst solution(s). Note that many of the solutions found by PCPE can have the same fitness value. Values closer to 0 in *St Dev* would indicate that most solutions are similar and their fitness value is similar to the mean fitness value. However, the mean *St Dev* is 1.21, showing that in general solutions are spread out, i.e., they are different when compared against each other, even though very little static information is provided to the PCPE algorithm (as shown in the column *Input Query* of Table 1). This fact is evident when we look at the fitness of the different solutions in a graphical way. In Fig. 2 we can observe, for the `nrev` benchmark, as defined in Listing 1(a), how the fitness of all solutions are quite distributed across the mean value. We have chosen this benchmark because it is the one with the lowest *Standard Deviation* value, and with the highest number of versions obtained. Also, we can see that many solutions share the same fitness value, and that in some way they are grouped together, indicating that it should be possible to find ways to collapse those solutions into one, pruning in this way the search space. Regarding the *Diameter* column, we can observe that the mean diameter is 3.83, indicating that there is an important difference between the worst and the best solutions.

These preliminary results are encouraging, showing that PCPE is capable of obtaining several heterogeneous solutions, most of them not being achievable by traditional partial evaluation. Similar results have been obtained for other fitness functions (not shown here due to lack of space). Though it is clear we need to prune the search space in order to make this approach practical, we should do it with care, in order to not to prune the good solutions.

5 Pruning the Search Space: SPRS Heuristic

In spite of the possibility of eliminating redundant configurations and non-promising branches, it is worthwhile to explore in practice the use of poly-controlled partial deduction with more restrictive capabilities in order to reduce the cost of exploring the search space. For instance, rather than allowing all possible combinations of specialization strategies for different atoms in a configuration, we can restrict ourselves to configurations which always use the same specialization strategy for all atoms which correspond to the same predicate. This restriction will often signifi-

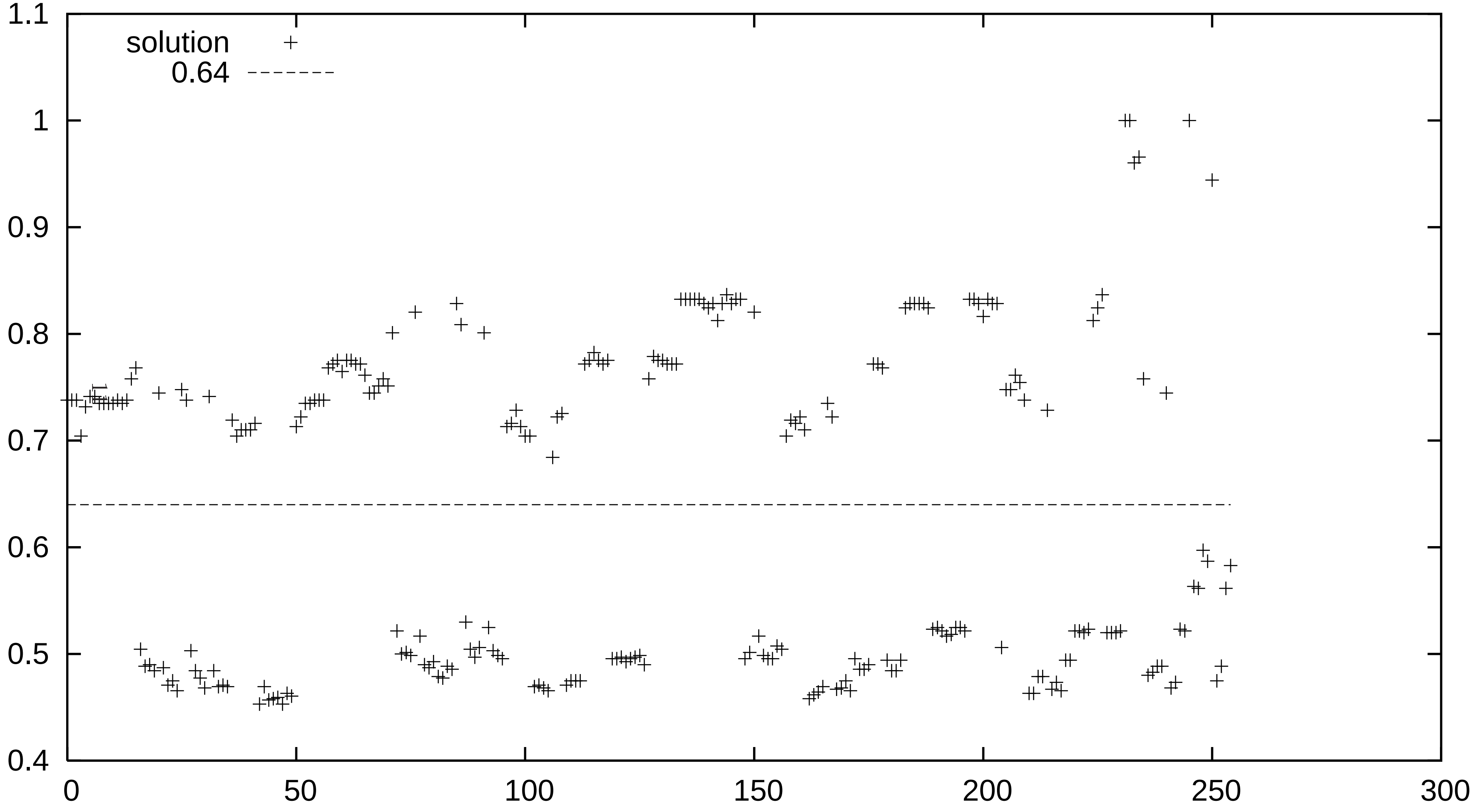

Fig. 2. PCPE solutions for nrev

cantly reduce the branching factor of our algorithm since, handling of an atom A_i will become deterministic as soon as we have previously considered an atom for the same predicate in any configuration which is an ancestor of the current one in the search space, i.e., it is compulsory to use exactly the same specialization strategy as before. We call this approach SPSR, standing for *Same Predicate, Same Rules*. We will refer to configurations which satisfy this restriction as *consistent*, and as *inconsistent* to those which do not. Though this simplification may look too restrictive at first sight, it is often the case in practice that there exists a specialization strategy which behaves well for all atoms which correspond to the same predicate, in the context of a given program.

We will modify Algorithm 1 in such a way that only consistent configurations are further processed. For this we need to store for every atom in every configuration the global control rule used to generalize such an atom. We now provide a formal definition of consistent configurations w.r.t. to the SPSR heuristic.

Definition 5.1 [consistent configuration] given a configuration $\mathit{Conf} = \langle S, H\rangle$, we say that Conf is *consistent* iff $\forall\langle A_1, A'_1, G_1, U_1\rangle \in H,\ \forall\langle A_2, A'_2, G_2, U_2\rangle \in H,\ \mathit{pred}(A_1) = \mathit{pred}(A_2) \Rightarrow (G_1 = G_2 \wedge U_1 = U_2)$

Note that the definition of consistent configuration can be applied to intermediate configurations (not only to final ones). Thus, if a given configuration *Conf* is inconsistent, it will be pruned, i.e., it will not be pushed on *Confs*. By doing this we are pruning not only this configuration, but also all the successor configurations that would have been generated from it. This means that early pruning will achieve significant reductions of the search space.

6 Experimental Results

Since the *SPSR* heuristic prunes the search space in a *blind* way, i.e., without making any evaluation of the candidates being pruned, there is a possibility of pruning the optimal solutions. In order to determine if this is the case, we have extended the experiments shown in Sec. 4, adding the results obtained when applying the *SPSR* heuristic to the example programs.

In Table 2, we show the number of *versions* obtained by PCPE, the *fitness* value of both the optimal solution(s) obtained by PCPE, and the best solution obtained by traditional PE (together with the control strategy *CS* used to obtain such value[3]), the *mean* value of all solutions, their *standard deviation* and their *diameter*, when using *speedup* as a fitness function. We compare in all cases the values obtained by the original PCPE approach (in row *orig* under colum *Heur*) versus the values obtained by PCPE when pruning its search space by means of the *SPSR* heuristics (in row *spsr*).

As shown in the table, the search space is significantly reduced when applying *SPSR*, and the *mean* number of versions is reduced from 87 candidate solutions to only 14. However, there are some benchmarks for which no pruning of the search space is achieved, as is the case of `example_pcpe` and `ssupply`. This is due to the

[3] We use the following notation for denoting pairs of control rules: ho={hom_emb,one_step}, hd={hom_emb,df_hom_emb_as}, do={dynamic,one_step}, dd={dynamic,df_hom_emb_as}

Benchmark	Heur	Versions	Fitness			Mean	St Dev	Diameter
			PCPE	CS	PE			
example_pcpe	orig	27	1.56	hd	1.37	0.87	0.21	0.99
	spsr	27	1.60			0.86	0.23	1.11
permute	orig	70	1.31	hd	1.06	0.91	0.48	1.16
	spsr	9	1.29			1.02	1.01	1.01
nrev	orig	255	1.03	hd	1.03	0.64	0.15	0.51
	spsr	9	1.06			0.71	0.19	0.55
advisor	orig	14	1.68	hd	1.55	1.21	0.67	0.97
	spsr	8	1.66			1.49	0.86	1.06
relative	orig	61	18.01	hd	15.30	3.45	4.84	16.37
	spsr	11	17.96			8.00	9.36	16.95
ssuply	orig	31	5.15	hd	5.15	1.52	1.82	4.72
	spsr	31	5.13			1.53	1.82	4.51
transpose	orig	154	2.62	hd	2.60	0.87	0.30	2.13
	spsr	6	2.54			1.08	0.57	1.60
overall	**orig**	**87.4**	**4.49**		**4.01**	**1.35**	**1.21**	**3.83**
	spsr	**14.4**	**4.44**			**2.09**	**2.01**	**3.82**

Table 2
Comparison of search-pruning alternatives(speedup)

fact that these programs contain very few atoms in their candidate specializations, and all of such configurations are consistent, satisfying the *SPSR* restriction.

In our experiments, when pruning is done, the *St Dev* grows, indicating that we are pruning solutions sharing the same fitness value. By looking at the *fitness* values, we can presume that the best solution is preserved, in spite of performing a blind pruning (the slight difference between fitness values of *orig* and *spsr* is probably due to noise when measuring time). Note that, in most cases, PCPE outperforms traditional PE. Interestingly, it is clear that for these benchmarks the best strategy for PE is *hd*. We can observe also that the mean fitness is higher when pruning is performed, which could indicate that bad solutions are pruned away.

In Table 3 we show the same information as above, but for the *reduction* fitness function. We have also added an extra column *Sols* showing the number of best solutions found by PCPE (note that this column does not make any sense when time-efficiency is measured, because this measurement is subject to noise). By looking at the fitness value, we can see that the best solution is preserved, in spite of performing a blind pruning. But according to the *Sols* column, we are pruning away the redundant best solutions, and leaving only one of them. Clearly, the number of versions pruned by *SPSR* does not depend on the fitness function used, since the fitness function is used *after* generating all solutions in order to determine which candidates are the best ones.

With regard to the fitness value, it is interesting to note that the strategy *do*, i.e., *dynamic* as a global control and *one_step* as a local control, produces a program that is very similar to the original one (probably having some variable and predicate renaming). This means that in situations where the original program has few predicates, it is difficult to obtain a residual program smaller than the original program. This is reflected in the benchmarks `permute`, `nrev`, `relative` and `transpose`, where the best control strategy is *do* and the fitness value is close to 1. However, note that PCPE still obtains better solutions in the cases of `permute` and

Benchmark	**Heur**	**Versions**	**Sols**	**Fitness**			**Mean**	**St Dev**	**Diameter**
				PCPE	**CS**	**PE**			
example_pcpe	orig	27	1	1.22	hd	1.15	0.82	0.19	0.82
	spsr	27	1	1.22			0.82	0.19	0.82
permute	orig	70	6	1.15	do	0.98	0.61	0.27	1.15
	spsr	9	1	1.15			0.63	0.34	1.15
nrev	orig	255	3	0.98	do	0.98	0.32	0.15	0.79
	spsr	9	1	0.98			0.55	0.25	0.71
advisor	orig	14	1	1.69	hd	1.68	1.03	0.34	1.41
	spsr	8	1	1.69			0.94	0.38	1.41
relative	orig	61	2	1.17	do	0.98	0.67	0.25	1.04
	spsr	11	1	1.17			0.80	0.28	1.04
ssuply	orig	31	1	11.26	hd	11.26	1.61	1.79	10.32
	spsr	31	1	11.26			1.61	1.79	10.32
transpose	orig	154	5	0.98	do	0.98	0.39	0.19	0.75
	spsr	6	1	0.98			0.63	0.26	0.70
overall	**orig**	**87.4**	**2.71**	**2.63**		**2.57**	**0.77**	**0.45**	**2.32**
	spsr	**14.4**	**1.00**	**2.63**			**0.85**	**0.49**	**2.30**

Table 3
Comparison of search-pruning alternatives(reduction)

`relative`, clearly through a hybrid solution.

It is also interesting to see that the diameter is preserved most of times, indicating that both the best and worst solutions are preserved. However, in `nrev` and `transpose` the diameter decreases a bit, and since the best solution is preserved, this means we are pruning the worst solutions in these cases.

In summary, *SPSR* seems to be a very interesting pruning technique, since it significantly reduces the search space of PCPE, it seems to preserve the best solutions (at least for the tested benchmarks), and can allow us to use PCPE in order to attack more interesting benchmarks, and also to provide more static information to the algorithm. It remains as future work to develop other techniques for pruning the search space in PCPE, that can ensure that the optimal solution is preserved.

Acknowledgments.

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-15905 *MOBIUS* project, by the Spanish Ministry of Education under the TIN-2005-09207 *MERIT* project, and by the Madrid Regional Government under the S-0505/TIC/0407 *PROMESAS* project.

References

[1] E. Albert, G. Puebla, and J. Gallagher. Non-Leftmost Unfolding in Partial Evaluation of Logic Programs with Impure Predicates. In *14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'05)*, number 3901 in LNCS. Springer-Verlag, April 2006.

[2] M. Bruynooghe, D. De Schreye, and B. Martens. A General Criterion for Avoiding Infinite Unfolding during Partial Deduction. *New Generation Computing*, 1(11):47–79, 1992.

[3] Stephen-John Craig and Michael Leuschel. Self-tuning resource aware specialisation for Prolog. In *PPDP '05: Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 23–34, New York, NY, USA, 2005. ACM Press.

[4] Saumya K. Debray. Resource-Bounded Partial Evaluation. In *Proceedings of PEPM'97, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 179–192. ACM Press, 1997.

[5] J. Gallagher and M. Bruynooghe. The derivation of an algorithm for program specialisation. *New Generation Computing*, 9(1991):305–333, 1991.

[6] J.P. Gallagher. Tutorial on specialisation of logic programs. In *Proceedings of PEPM'93, the ACM Sigplan Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98. ACM Press, 1993.

[7] Manuel V. Hermenegildo, Germán Puebla, Francisco Bueno, and Pedro López-García. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming*, 58(1–2):115–140, October 2005.

[8] M. Leuschel and M. Bruynooghe. Logic program specialisation through partial deduction: Control issues. *Theory and Practice of Logic Programming*, 2(4 & 5):461–515, July & September 2002.

[9] Michael Leuschel. On the power of homeomorphic embedding for online termination. In Giorgio Levi, editor, Static Analysis. *Proceedings of SAS'98*, LNCS 1503, pages 230–245, Pisa, Italy, September 1998. Springer-Verlag.

[10] Michael Leuschel, Bern Martens, and Danny De Schreye. Controlling generalisation and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, January 1998.

[11] J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *The Journal of Logic Programming*, 11:217–242, 1991.

[12] J.W. Lloyd. *Foundations of Logic Programming*. Springer, second, extended edition, 1987.

[13] B. Martens and D. De Schreye. Automatic finite unfolding using well-founded measures. *Journal of Logic Programming*, 28(2):89–146, 1996. To Appear, abridged and revised version of Technical Report CW180, Departement Computerwetenschappen, K.U.Leuven, October 1993.

[14] G. Puebla, E. Albert, and M. Hermenegildo. Efficient Local Unfolding with Ancestor Stacks for Full Prolog. In *14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04)*, number 3573 in LNCS, pages 149–165. Springer-Verlag, 2005.

[15] G. Puebla and C. Ochoa. Poly-Controlled Partial Evaluation. In *Proc. of 8th ACM-SIGPLAN International Symposium on Principles and Practice of Declarative Programming (PPDP'06)*. ACM Press, July 2006.

[16] M.H. Sørensen and R. Glück. An Algorithm of Generalization in Positive Supercompilation. In *Proc. of ILPS'95*, pages 465–479. The MIT Press, 1995.