Probabilistic Procedure Cloning for High-Performance Systems *†

Siegfried Benkner[1] Eduard Mehofer[1] Bernhard Scholz[2]

[1] Institute for Software Science
University of Vienna
Vienna – Austria
sigi@par.univie.ac.at, mehofer@par.univie.ac.at

[2] Institute for Computer Languages
Vienna University of Technology
Vienna – Austria
scholz@complang.tuwien.ac.at

Abstract—

For many scientific and engineering applications efficient compilation of Fortran 90 code is of paramount importance in order to exploit computing power of high performance systems. Fortran 90 subroutine calls can cause serious performance losses if copy-in/copy-out argument passing has to be applied. We present an interprocedural cloning algorithm that prevents redundant argument copying. Our approach is novel with respect to utilizing probabilistic data-flow analysis, which allows us to identify profitable procedure clones, to control code growth by a parameterizable threshold, and to create multiple calls of procedure clones even for single call-sites. On high-performance systems experimental results illustrate the importance and effectivity of this kind of optimization.

Keywords— **Procedure Cloning, Fortran 90, Probabilistic Data-Flow Analysis.**

I. Introduction

The vast majority of scientific and engineering applications is written in Fortran. In recent years several standards for the development of parallel applications on high performance computing systems in Fortran have emerged such as data-parallel languages like HPF [6] and language extensions like MPI [7] and OpenMP [2]. For such applications efficient compilation of Fortran 90 is of paramount importance. In this paper we present a novel procedure cloning algorithm which was motivated by a Fortran 90 problem that can cause serious performance losses. Particularly this problem may occur when Fortran 90 code is merged with existing Fortran 77 code. While in Fortran 77 the principal mechanism of passing array arguments is *call-by-reference*, in Fortran 90 a copy-in/copy-out argument transfer strategy may have to be adopted. In Fortran 90 the storage of explicit shape arrays and assumed shape arrays may differ. Whereas for explicit shape arrays a contiguous memory storage is required, assumed shape arrays may also be stored in non-contiguous memory areas. The different storage schemes may influence the parameter passing mechanism. An actual argument which is not contiguous and which is passed to a contiguous dummy array has to be copied in a contiguous temporary array on entry of a procedure and copied out at procedure exit. If a compile cannot statically determine whether an actual array argument is contiguous or not, copy-in/copy-out is usually applied which may crucially increase the calling overhead. Copy-in/copy-out may occur in the following cases (see Figure 1) :

(a) An *explicit shape* dummy array `X` requiring a contiguous memory storage (also called adjustable array in F77 terminology) is associated with a non-contiguous section of an actual array `A`.
(b) An *explicit shape* dummy array `X` requiring a contiguous memory storage is associated with an actual array `PA` that has the `POINTER` attribute and `PA` has been pointer-associated with a non-contiguous array section.
(c) An *explicit shape* dummy array `X` requiring a contiguous memory storage is associated with an actual array `DA` that is declared as an assumed shape array in the scope of the caller which need not to be contiguous.

In order to avoid a copy-in/copy-out argument transfer procedure cloning can be applied to replace explicit shape dummy arrays by assumed shape dummy arrays. However, a main problem of procedure cloning is the possibility of code explosion, which can be in the worst case exponential [8]. We tackle this problem in our approach by employing a probabilistic data flow system. Based on the solution of a probabilistic data-flow analysis problem the probability of an actual argument to be contiguous or not is calculated and a ranking of potential procedure signatures is determined at

*This research is partially supported by the Austrian Science Fund as part of Aurora Projects “Languages and Compilers for Scientific Computation” and “Tools” under Contract SFB-011.
†Accepted for publicatin at 12th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD’2000), Sao Pedro, Brazil, October 2000.

a call-site. A threshold controls program growth by selecting only those signatures for procedure cloning which are above the threshold.

```
subroutine sub(..., DA, ...)
    ! assumed shape array
   double precision          :: DA(:)
    ! local array of sub()
   double precision, target  :: A(na)
    ! pointer to a 1-D array section
   double precision, pointer :: PA(:)
   ...
    ! case (a)
   call sub1(A(l:u:s),size(A(l:u:s)))
   ...
    ! case (b)
   PA => A(l:u:s); ...
   call sub1(PA, size(PA))
   ...
    ! case (c)
   call sub1(DA, size(DA))
   ...
end subroutine sub

subroutine sub1(X, xn)
   ! explicit shape dummy array
   double precision :: X(xn)
   integer xn
   ...
end subroutine sub1
```

Fig. 1. Explicit Shape Dummy Arrays

In this paper we present a novel interprocedural cloning algorithm that exhibits following characteristics:

- Runtime information is used to identify profitable clones.
- Code growth vs. performance gain is parameterizable.
- An original procedure call may be replaced by a sequence of guarded calls to procedure clones.
- Our framework is generic and easily adaptable to other cloning problems as well.

Our paper is organized as follows. In Section II we outline the basic transformation idea in the Fortran 90 context and motivate a probabilistic approach. In Section III we introduce basic notions used in this paper. Section IV presents our probabilistic procedure cloning algorithm. Our experiments are summarized in Section V. Related work is surveyed in Section VI and, finally, we draw our conclusions in Section VII.

II. Basic Transformation

For the example code of Figure 1, case (b), our procedure cloning transformation is sketched in Figure 2. A clone `sub1_cloned()` of subroutine `sub()` is generated, where the original explicit shape dummy array `X` is replaced by an assumed shape dummy array. Since an assumed shape array does not need to occupy a contiguous memory area, argument copying can be avoided in the case where the actual array corresponds to a contiguous array section. The original call-site is transformed by introducing an `if`-statement to ensure that the cloned subroutine is called in case of a contiguous actual argument only. If a non-contiguous array section is passed as argument (i.e. `s` $\neq$ `1`), the original subroutine is called and a copy-in/copy-out semantics may have to be adopted by the F90 compiler as usual.

```
...
PA => A(l:u:s)
...
if (PA is a contiguous array section) then
   ! no copy
   call sub1_cloned(PA, size(PA))
else
   ! copy will be generated by compiler
   call sub1(PA, size(PA))
end if
...
subroutine sub1_cloned(X, nx)
   ! assumed shape dummy array
   double precision, dimension(:) :: X
   ...
end subroutine sub1_cloned
```

Fig. 2. Optimization of argument transfer based on probabilistic procedure cloning for example shown in Fig. 1, case (b) .

As mentioned above, a main issue for procedure cloning is to prevent exponential code growth. The example code in Figure 3 shows a subroutine with four array arguments. Since we have for each array argument the choice between call-by-reference and copy-in/copy-out argument passing, there exist potentially 2^4 clones of subroutine *sub*. In order to prevent code explosion we have to restrict the number of clones which ought to be generated. In our example pointer assignment analysis is the key to determine profitable clones. Note that classical data-flow analysis yields that *PArr3* may refer to non-contiguous array section $A(::2)$ or to contiguous array B. Unfortunately, this information is too coarse-grained to make a decision whether a procedure clone should be generated to prevent copying. Probabilistic data-flow analysis, however, gives us exactly the information what is needed to solve that problem. Instead of the information whether a contiguous memory layout may reach or does not reach a call-site, the probability is computed with what a contiguous or non-contiguous memory layout will reach a call-site. Assume that the probability for *PArr3* to be non-contiguous (i.e. refers to $A(::2)$) is 90%, and to be contiguous (i.e. refers to B) is only 10%. Hence, it could be desirable to discard a contiguous specialization for argument *PArr3*, since its probability of 10% is too low.

The example shown in Figure 3 reveals an interesting property of our cloning approach. Whereas in related work procedure cloning is primarily based on different call-sites, we perform procedure cloning even for one call-site by replacing the orginal call by a sequence of guarded calls to the most profitable procedure clones.

```
...
     PArr3 => A(::2)
...
  PArr3 => B
.
.   ! more pointer assignments
.
call sub(PArr1,size(PArr1),PArr2,size(PArr2),
         PArr3,size(PArr3),PArr4,size(PArr4))
:::
subroutine sub(Y1,ny1,Y2,ny2,Y3,ny3,Y4,ny4)
   ! explicit shape dummy array
   double precision :: Y1(ny1),Y2(ny2)
   double precision :: Y3(ny3),Y4(ny4)
   ...
end subroutine sub
...
```

Fig. 3. Code explosion.

III. Preliminaries

A program consists of a collection of control flow graphs, one of which, G_{Main}, represents the program's main procedure. Each flow graph G_p of procedure p is a *directed graph* with node set N_p and edge set $E_p \subseteq N_p \times N_p$. Edges $x \rightarrow y \in E_p$ represent statements and model the nondeterministic branching structure of G_p. Each flow graph G_p has a *start node* s_p and *end node* e_p. The set of all *immediate predecessors* of a node y of G_p is $pred(y) = \{\, x \mid x \rightarrow y \in E_p \,\}$. Let $CS_p \subseteq E_p$ be the set of call-sites of procedure p. For a call-site $cs \in CS_p$, $ProcName(cs)$ denotes the procedures called at cs.

As usual, a monotonic intraprocedural data-flow analysis problem is a tuple $DFA = (L, \wedge, F, c, G_p, M)$, where L is a bounded semilattice with meet operation $\wedge$, $F \subseteq L \rightarrow L$ is a monotone function space associated with L, $c \in L$ are the "data-flow facts" associated with start node s_p, and $M : E_p \rightarrow F$ is a map from G_p's edges to data-flow functions.

For bitvector problems the semilattice L is a powerset 2^D of finite set D. Bi-distributive bitvector problems require that all functions in function space F distribute over both set union and set intersection.

A probabilistic data flow analysis(PDFA) problem is a tuple (*DFA*,*RT*) where *DFA* is a bi-distributive bitvector problem and *RT* represents runtime information obtained by profiling. Let $E(x, \Lambda)$ denote the number of times a node x is expected to be executed, and $E(x, d)$ denote the expected number of times that d hold true at program point x. A PDFA framework computes $E(x, \Lambda)$ and $E(x, d)$.

In this paper actual and dummy argument types are represented by type vectors $\vec{t} \in T^l$ of length l where T is the set of possible types (e.g., contiguous and non-contiguous). A signature of a procedure p is a tuple $(p, \vec{t})$ which comprises procedure name and dummy argument types $\vec{t}$.

IV. Probabilistic Procedure Cloning

Our interprocedural framework consists of (1) a PDFA framework, which computes probabilities of occurrences for argument types, (2) a clone selector, which decides which argument types are profitable with respect to performance and code size, (3) a code transformer, which performs the code changes to the original code on the call-site and clones the procedure with appropriate sets of dummy argument types.

The probabilistic data flow analysis is an intraprocedural analysis. For a procedure clone with a given signature a PDFA framework computes types of actual arguments (e.g., contiguous array shape or non-contiguous array shape) at call sites. Based on this information new procedure clones are determined and further analyzed. An interprocedural fixpoint algorithm drives the whole process as long as all profitable clones are computed and generated.

A. PDFA Framework

Conventional data flow analysis computes information about what facts may or will not hold during the execution of a program. In contrary probabilistic data flow analysis (PDFA) computes probabilities of data flow facts (cf. [5, 9]). For our procedure cloning algorithm we employ a PDFA framework to calculate probabilities of argument types (e.g., contiguous and non-contiguous) for a procedure call. To reduce problem-space we only consider those variables which are directly or indirectly involved at call-sites (e.g., actual arguments) since other variables do not affect the copy-in/copy-out argument transfer mechanism of Fortran 90. More formally, the set of variables is denoted by $\{v_1, \ldots, v_k\}$, and the set of *types* is given by $T = \{t_1, \ldots, t_n\}$. A power-set 2^D is used to represent possible type bindings for variables, where

$$D = \underbrace{\{d_{t_1}^{(v_1)}, \ldots, d_{t_n}^{(v_1)}\} \cup \ldots \{d_{t_1}^{(v_k)}, \ldots, d_{t_n}^{(v_k)}\}}_{k}$$

with $d_{t_i}^{(v_j)}$ denoting that variable $v_j \in V$ has type $t_i \in T$. For each node $y \in N_p$, $\mathbf{Tp}(y) \in 2^D$ comprises the type bindings for all variables in node y.

In the sequel we define two (local) functions **TKill** and **TDefs** from the set of edges E_p to the power set of D. Function **TKill**$(x \rightarrow y)$ computes the set of data-flow facts $d_{t_i}^{(v_j)}$ that are killed in edge $x \rightarrow y$. A data-flow fact $d_{t_i}^{(v_j)}$ is member of **TKill**$(x \rightarrow y)$ iff variable v_j is assigned a new type within edge $x \rightarrow y$. Function **TDefs**$(x \rightarrow y)$ computes type changes for variables at edge $x \rightarrow y$. If variable v_j is assigned a new type t_i within edge $x \rightarrow y$ and the type of variable v_i is not subsequently modified within edge $x \rightarrow y$, then $d_{t_i}^{(v_j)}$ is element of **TDefs**$(x \rightarrow y)$.

The fundamental relationship to compute probabilities of type bindings for variables is given in Figure 4.

TP1. $\mathbf{Tp}(s_p) = c(\vec{t})$.

TP2. For each node y, $y \neq s$,

$$\mathbf{Tp}(y) = \bigcup_{x \in pred(y)} ([\mathbf{Tp}(x) - \mathbf{TKill}(x \to y)] \cup \mathbf{TDefs}(x \to y)).$$

Fig. 4. Equations for "type propagation"

For node s_p, $\mathbf{Tp}(s_p)$ is assigned the value of $c(\vec{t}) \in 2^D$ where $c(\vec{t})$ is a function which maps the signature of the clone to a type binding for variables.

Based on the runtime information, the PDFA framework [5] calculates probabilities of occurrences for argument types of a procedure. For all program points $y \in N$ expected frequencies $E(y, d_{t_i}^{(v_j)})$ of data facts and expected frequencies $E(y, \Lambda)$ of nodes are calculated. In more detail, the probability of type t_i for variable v_j that may reach point y is given as follows,

$$P(y, d_{t_i}^{(v_j)}) = \begin{cases} \frac{E(y, d_{t_i}^{(v_j)})}{E(y,\Lambda)}, & \text{iff } E(y, \Lambda) \neq 0 and \\ 0, & \text{otherwise.} \end{cases}$$

B. Clone Selector

The clone selector uses the most promising *cloning candidates* for cloning at call-site *cs*. A cloning candidate represents actual argument types at *cs*. Let $\mathit{CVars}(cs) = \{v_{i_1}, \dots, v_{i_m}\} \subseteq V$ be the variables that are taken as actual arguments at call-site *cs*. The possible actual argument types at call-site *cs* are given by the m-th cross-product over the type domain T,

$$\vec{t} = < t_1^{(v_{i_1})}, t_2^{(v_{i_2})}, \dots, t_m^{(v_{i_m})} > \in \underbrace{T \times T \dots \times T}_{m},$$

where $m = |\mathit{CVars}(cs)|$. Under the assumption that occurrences of argument types are independent of each other, the probability is given as follows[1]

$$P(x, < t_1^{(v_{i_1})}, t_2^{(v_{i_2})}, \dots, t_m^{(v_{i_m})} >) = P(x, d_{t_1}^{(v_{i_1})}) \cdot P(x, d_{t_2}^{(v_{i_2})}) \cdot \dots \cdot P(x, d_{t_m}^{(v_{i_m})}),$$

where x is the source of call-site $cs = x \to y$.

We define a threshold *th* in a range between zero and one. Only those types of actual arguments are taken for cloning, that are greater than the threshold to the power of $m = |\mathit{CVars}(cs)|$. This threshold parameterizes the cloning algorithm. A threshold of zero enables all possible argument types at call-site *cs* and we can expect exponential code growth. In contrary a threshold of one allows only few argument types with the probability of one. More formally, the set of cloning candidates is given below,

$$\mathit{Cand}(cs) = \{\vec{t} \,|\, P(cs, \vec{t}) \geq th^m\}.$$

If set $\mathit{Cand}(cs)$ is empty, we choose default argument types. We can either take the most likely ones or the dummy types that the programmer has originally taken for the called procedure.

[1] If this assumption is violated, our cloning transformation is still semantically correct. However, not the most profitable clones could be selected.

```
procedure InterproceduralFramework
    Worklist := {(Main, 0⃗)}
    Analysed := ∅;
    while Worklist ≠ ∅
        NewProcs := ∅;
        foreach Proc in Worklist
            NewProcs := NewProcs ∪
                Cloning( Proc_Name, Proc_Signature );
            Analysed := Analysed ∪ { Proc };
        end for
        Worklist := (Worklist ∪ NewProcs) - Analysed;
    end while
end procedure
```

Fig. 5. Interprocedural Framework

C. Interprocedural Cloning Algorithm

The interprocedural cloning algorithm propagates argument types to the callees as described in Figure 5.

Clones which have to be analyzed are stored in a *Worklist*. The *Worklist* stores signatures which comprise procedure name and corresponding dummmy argument types, and which denote either original procedures or clones which shall be analyzed and generated. At the beginning the *Worklist* is initialized with the signature of the main procedure where $\vec{0}$ denotes that there are no relevant dummy arguments of the main procedure for procedure cloning. A second set, called *Analyzed*, contains all signatures which have already been analyzed and is initialized with the empty set.

In the inner for-loop of the algorithm we perform probabilistic *Cloning* for each signature in the *Worklist* and add thereafter the clone to *Analyzed*. All clones which result from an iteration over the *Worklist* as done by the inner for-loop are stored in *NewProcs*. By adding *NewProcs* to the *Worklist* the dummy argument types are interprocedurally propagated through the call graph. Of course, signatures which

```
function Cloning(p, t⃗)
    NewClones := ∅ ;
    // perform probabilistic data flow analysis of
    // procedure p and type vector t⃗
    PerformPDFA(p, t⃗);
    // generate clone with appropriate header(t⃗)
    // without call-sites
    GenerateClone(p, t⃗);
    // specialize call-sites and memorize specialized
    // call-sites in NewClones
    foreach callsite cs in CS_p:
        /* get cloning candidates */
        Cand:=Cand(cs) = { t⃗ | P(cs, t⃗) ≥ th^m };
        /* insert code for call-site cs */
        SpecializeCallSite(cs,Cand);
        foreach c⃗ in Cand:
            NewClones:=NewClones ∪
                { (ProcName(cs), ParamMap(cs, c⃗)) };
        endfor
    endfor
    return NewClones;
endfunction
```

Fig. 6. Cloning algorithm

have already been analyzed need not to be dealt with a second time and are subtracted from the *Worklist* before in the outer while-loop it is tested whether the *Worklist* is empty or still some work has to be done. Our algorithm is a fixed-point algorithm and, therefore, can handle recursions as well.

It can be easily seen that the algorithm will terminate since the number of different types is finite, and the number of signatures is finite as well. The *Analyzed* set is monotonically growing and once no new signatures will be generated which have not already been dealt with resulting in an empty *Worklist* and the termination of the algorithm.

D. Intraprocedural Cloning Algorithm

The intraprocedural cloning algorithm in Figure 6 fulfills two tasks. First, the code of clone p with dummy type arguments $\vec{t}$ is generated. Second, the promising procedure clones for each call-site within clone p are determined.

The algorithm in Figure 6 consists of three major steps. First, the probabilistic data flow analysis (see Section A) is performed by PerformPDFA$(p, \vec{t})$. Second, the code of clone p with dummy type arguments $\vec{t}$ is generated by GenerateClone$(p, \vec{t})$. A unique name is given to the clone and the dummy arguments of the clone are specialized according to dummy argument types $\vec{t}$. Note that the code of all call-sites is left out and inserted later. Third, the code of call-sites is inserted and result set *NewClones* is computed. In this section of the algorithm a for-loop analyzes all call-sites of the clone. Inside the loop all cloning candidates of call-site *cs* are determined as described in Section B. Thereafter, the code of the call-site is generated by replacing the call of the original procedure p by the code transformation template as outlined in Figure 7.

```
! Generate code for cloning candidate t⃗
if ((stype(v_i1).eq.t_1^(v_i1)).and.
 ... (stype(v_im).eq.t_m^(v_im))) then
   call ProcName(cs)_ParamMap(cs,t⃗) (args ...)
else
   ! Generate code either for another cloning
   ! candidate or for default clone
   ....
end if
```

Fig. 7. Code transformation template

Note that function `stype(`v`)` returns the type of variable v and is either implemented by the runtime library or an instrumentation is to be implemented to obtain this information. Furthermore, the called clone is denoted by *ProcName*$(cs)_{ParamMap(cs,\vec{t})}$ where *ProcName*(cs) determines the procedure of call-site *cs* and *ParamMap* maps the actual argument types to the dummy arguments of the clone. In the last step of the loop body cloning candidates are converted to dummy argument types and added to result set *NewClones*.

V. Experiments

In Table I we show the performance of a kernel which performs a 5-point stencil on a one-dimensional array. The kernel was measured on a single vector processor (8 GFLOPS) of an NEC SX-5 with NEC F90/SX compiler and on a single processor (250 MHz) of an SGI Origin 2000 with MIPSpro Fortran 90 compiler (version 7.3.1). The kernel performs 1000 time-steps, where in each time-step a subroutine is called to perform the stencil operation. This subroutine takes a pointer to an array section as actual argument. In our experiments the stride of the transferred array section, although specified as a variable computed at runtime, was always 1.

The first entry in the table corresponds to the time required for the original code where the dummy array of the stencil procedure is declared as an explicit shape array, resulting in a copy-in/copy-out argument transfer. The second entry corresponds to a variant of the kernel optimized by means of our probabilistic data-flow framework. In this version the argument transfer is optimized by calling a clone of the stencil procedure which utilizes an assumed shape dummy array as described in Figure 2. As a consequence, in the optimized

Machines	Code version	1 K	10 K	100 K	1000 K
NEC SX-5	original code (explicit shape dummy)	0.005 s	0.019 s	0.17 s	1.74 s
	optimized code (assumed shape dummy)	0.006 s	0.015 s	0.11 s	1.27 s
SGI Origin 2000	original code (explicit shape dummy)	0.078 s	1.02 s	10.64 s	190.4 s
	optimized code (assumed shape dummy)	0.063 s	0.79 s	8.41 s	129.0 s

TABLE I: Optimizing the transfer of array sections by means of procedure cloning. The numbers in the table are the elapsed times (sec) for a computational kernel performing 1000 iterations, where in each iteration a procedure to perform a 1-D 5-point stencil operation is called. In our experiments `double precision` arrays with a size ranging from 1K up to 1000K elements were used. The second row of the table refers to the timings obtained where a pointer to an array section is passed to an explicit shape dummy array, while the last row shows the elapsed times where a cloned variant of the procedure with an assumed dummy shape array is called (see Figure 1 (a)).

code version the array argument is passed by reference. The huge differences in the elapsed times between the two versions is caused by the overhead of copying the actual array section on entry to the procedure as well as upon exit from the procedure.

Although the experiments shown in Table I indicate a significant performance improvement for the case where explicit shape arrays are substituted by assumed shape arrays, such a transformation may also cause certain overheads with some Fortran compilers. When computing linear adresses for accesses to assumed arrays, a Fortran compiler has to take the information about the transferred actual array section (i.e. the stride) into account. The stride information is usually passed in addtion to the address of the actual array by means of a section descriptor in the array's dope vector. The stride of the actual array section has to be incorporated into the address translation mechanism. In case of non-contiguous actual array section, additional multiplications may have to be performed during computation of linear addresses. Moreover, if the stride of the transferred array section is sufficiently large additional cache misses may be caused. As a consequence, certain overheads may be introduced for accessing elements of assumed shape arrays. Fortunatley, most Fortran 90 compilers can handle assumed shape arrays which are associated with stride 1 array sections as efficiently as explicit shape arrays.

VI. Related Work

Procedure cloning has been introduced by Cooper [3] in the context of interprocedural constants. Hall [8] and Cooper, Hall, and Kennedy [4] describe a procedure cloning algorithm which is based on three phases. In the first phase all cloning possibilities are explored by calculating cloning vectors which can be thought as a vector of information representing data facts used as basis for cloning. In the second phase equivalent cloning vectors are merged. Finally in the third phase procedure cloning is performed. Whereas in their approach cloning is restricted by identifying equivalent cloning vectors which produce identical optimization results, we use probabilistic data-flow information instead to restrict cloning to profitable ones. Furthermore, an original procedure call is replaced by a call to the cloned procedure. Our call-site transformation replaces the original call by a sequence of guarded calls to the most profitable procedure clones.

Probabilistic data-flow frameworks have been introduced by Ramalingam [9] with the description of a generic data flow framework which computes the probability that a data flow fact holds or does not hold for finite bi-distributive subset problems. The framework is based on the exploded control-flow graph introduced by Reps, Horwitz, Sagiv [10] and on Markov chains. However, execution history is not taken into account. We have modified the equation system as presented in [5] in order to utilize execution history. In this way we achieve significantly better results.

VII. Conclusion and Future Work

We have presented a novel interprocedural cloning algorithm that utilizes runtime information. In this way we can distinguish between profitable and non-profitable procedure clones with a high accuracy. A parameterizable threshold controls program growth. For a given threshold the most profitable procedure clones are created leading to the best performance possible based on our cost model. Contrary to related work, our cloning algorithm operates on single call-sites resulting in a replacement of an original procedure call by several clone calls. Although our cloning algorithm was motivated by the Fortran 90 argument passing problem, it is applicable for similar subtype propagation problems in other fields as well. The implementation of our approach is done in the context of VFC [1].

Currently we investigate several improvements of our procedure cloning algorithm. First, we would like to modify our algorithm such that clones of a procedure which result in the same PDFA problem are identified as identical and not re-analyzed again. Second, the transformation of a call-site could consider also clones generated at other call-sites for the same procedure.

REFERENCES

[1] S. Benkner. VFC: The Vienna Fortran Compiler. *Journal of Scientific Programming*, 7(1):67–81, December 1999.

[2] OpenMP Architecture Review Board. *OpenMP Application Program Interface*, November 1999. OpenMP Forum Home Page http://www.openmp.org/.

[3] K.D. Cooper. *Interprocedural data flow analysis in a programming environment*. PhD thesis, Rice University, Houston,TX, April 1983.

[4] K.D. Cooper, M.W. Hall, and K. Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2), April 1993.

[5] B. Scholz E. Mehofer. Probabilistic data flow system with two-edge profiling. In *Proceedings of the ACM SIGPLAN Workshop on Dynamic and Adaptive Compilation and Omptimization (Dynamo'00)*, Boston, MA, January 2000.

[6] High Performance Fortran Forum. High Performance Fortran language specification version 2.0. Technical report, Rice University, Houston,TX, January 1997. Available via HPFF home page: http://www.crpc.rice.edu/HPFF.

[7] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, July 1997. MPI Forum Home Page http://www.mpi-forum.org/.

[8] M. Hall. *Managing Interprocedural Optimization*. PhD thesis, Rice University, Houston,TX, April 1991.

[9] G. Ramalingam. Data flow frequency analysis. In *Proc. of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI'96)*, pages 267–277, Philadephia, Pennsylvania, May 1996.

[10] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. of the ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 49–61, San Francisco, CA, January 1995.