Removing Useless Remappings

—

Report A–275–CRI

Fabien COELHO
Corinne ANCOURT *

Centre de Recherche en Informatique, École des mines de Paris,
35, rue Saint-Honoré, 77305 Fontainebleau Cedex, FRANCE.
Phone: +33 1 64 69 47 08. Fax: + 33 1 64 69 47 09.
URL: `http://www.cri.ensmp.fr/pips`
FTP: `anonymous@ftp.cri.ensmp.fr`

October 23, 1995

Abstract

This report describes the optimization implemented in HPFC, our prototype HPF compiler, to remove useless remappings. This optimization is performed on the remapping graph. The construction of this graph from the control flow graph is first presented. Then the optimization is formalized as a may data-flow problem on this graph. Finally the results are discussed.

1 Remapping Graph

Useless remappings may appear naturally in HPF programs. First, the change of both alignment and distribution of an array requires a `realign` and a `redistribute`, hence resulting in two remappings if no special care is taken. Second, the redistribution of a template[1] induces the remapping of all aligned arrays, even if they are not all referenced afterwards. Third, at an interprocedural level, two consecutive subroutine calls may require the same remapping for a given array, resulting in a useless remapping on return from the first subroutine and on entry in the second. If two different mappings are required, it may also be interesting to remap data directly rather than using the intermediate original mapping. Such examples do not arise from badly written programs, but from a normal use of HPF features. They demonstrate the need for compile time optimizations to avoid useless costly remappings at runtime.

Let us consider example `remaps` in Figure 1. The loop nest involving two remappings is typical of ADI computations. Template `T` is redistributed at `1`, inducing `B`

* `{coelho,ancourt}@cri.ensmp.fr`
[1] Even when no templates are used [3] array redistributions generate the problem

and C remappings, but C is not referenced afterwards. Moreover argument A is never referenced with its initial mapping.

In this section, the remapping graph, its construction from the control flow graph and its optimizations are presented. This approach deals with descriptive and prescriptive mappings, *i.e.* when the compiler is aware of data distributions.

1.1 Construction

Let us introduce the remapping graph $\mathcal{G}_R$. This graph is a (usually much smaller) subgraph of the control flow graph. The vertices of the graph are the (re)mapping statements, *i.e.* `(re)align`s and `(re)distribute`s. An edge denotes a possible path in the control flow graph where a same array is remapped at both vertices. Two vertices are added at entry in and on exit from the subroutine. Mappings are designed by a number for each array. The same mapping number is used for identical mappings of an array. To each vertex v in $\mathcal{G}_R$ are associated:

- the set of remapped arrays $\mathrm{S}(v)$
- for all remapped arrays A in this set:
  - one leaving mapping[2] for the statement: $\mathrm{L}_{\mathtt{A}}(v)$
  - the set of mappings for A that may reach v: $\mathrm{R}_{\mathtt{A}}(v)$
  - whether it may be referenced after the remapping: $\mathrm{Used}_{\mathtt{A}}(v)$

This information is depicted in Figure 2. To the vertex is associated the remapped arrays A and B, with the leaving mapping as a subscript (0 for A, 1 for B) and the set of reaching mappings as a superscript ($\{1,3\}$ for A, $\{2\}$ for B). Referenced arrays are underlined (here only B). The compiler must generate remapping codes for each pair (reaching to leaving mappings).

Let us describe how $\mathcal{G}_R$ is built from the program control flow graph. First, the entry and exit vertices are created. The distributed subroutine local variables are attached to the entry, with their initial mapping as a leaving mapping. The subroutine distributed formal parameters are attached to both entry and exit vertices. The leaving mapping for those variables on entry is the initial mapping. The reaching (resp. leaving) mapping for the entry (resp. exit) is an *a priori* unknown X mapping. If the directives are descriptive, the reaching mapping on entry of the subroutine is the initial mapping (X=0). On exit, distributed formal parameters are tagged as used: without further interprocedural information, the compiler assumes that the final remapping is needed. Figure 3 shows the initial graph for `remaps`.

The next phase of the $\mathcal{G}_R$ construction is the propagation of the initial mappings from the subroutine entry, till remappings are encountered. This must be done for each couple $(v, \mathtt{A})$ of vertex and arrays remapped at this vertex. First, initialize the set of couples to be propagated with the entry vertex associated to the distributed arrays. Then for each such couple $(v, \mathtt{A})$, propagate in the control graph from the corresponding vertex till meeting remapping statements for that array mapping. Tag the array remapping as used if a reference is encountered while propagating. Let w

[2] several may occur, this assumption just simplifies the presentation

be one of the encountered remapping statements. Add a corresponding w vertex in $\mathcal{G}_R$ if necessary. Add $\mathtt{A}$ to $\mathrm{S}(w)$ and compute $\mathrm{L}_{\mathtt{A}}(w)$ if necessary, and $(w, \mathtt{A})$ is a new couple to be explored later on. Add $\mathrm{L}_{\mathtt{A}}(v)$ to $\mathrm{R}_{\mathtt{A}}(w)$. The resulting remapping graph for `remaps` is shown in Figure 4.

If n is the number of vertices in the control graph, s the maximum number of successors of a vertex, m the number of remapping statements and p the number of distributed arrays, then the worst case complexity of the outlined construction algorithm is $\mathcal{O}(nsmp)$, if all arrays are remapped at each remapping statements and the propagations in the control graph get through all vertices.

1.2 Optimization

In $\mathcal{G}_R$, arrays that are remapped after a remapping without having been referenced are tagged as not used for this remapping. In such cases, at least two remappings will be performed at runtime without referencing the array in between, as array `A` in Figure 4 after the entry vertex. Such useless remappings must be removed. However the successive remapping statements must be aware that they were not performed and that they may have to deal with other reaching mappings. Indeed, the whole set of reaching mappings must be recomputed. Some are no longer of use and others must be added. This optimization is performed as follow:

- First, remove all useless remappings[3], simply by deleting the leaving mapping for those vertices and arrays.

$$\forall v, \forall \mathtt{A} \in \mathrm{S}(v), \text{not } \mathrm{Used}_{\mathtt{A}}(v) \Rightarrow \mathrm{L}_{\mathtt{A}}(v) = \emptyset$$

- Second, recompute the mappings that may reach each vertex. This is a forward may data flow problem [2, 1] on $\mathcal{G}_R$:
  - initialization: Used 1-step reaching mappings

$$\forall v, \forall \mathtt{A} \in \mathrm{S}(v), \mathrm{R}_{\mathtt{A}}(v) = \bigcup_{\substack{w \in \mathrm{pred}(v) \\ \mathtt{A} \in \mathrm{S}(w),\ \mathrm{Used}_{\mathtt{A}}(w)}} \mathrm{L}_{\mathtt{A}}(w)$$

  - optimizing function: propagation

$$\forall v, \forall \mathtt{A} \in \mathrm{S}(v), \mathrm{R}_{\mathtt{A}}(v) = \mathrm{R}_{\mathtt{A}}(v) \cup \bigcup_{\substack{w \in \mathrm{pred}(v) \\ \mathtt{A} \in \mathrm{S}(w),\ \text{not } \mathrm{Used}_{\mathtt{A}}(w)}} \mathrm{R}_{\mathtt{A}}(w)$$

  The iterative resolution of the optimizing function is increasing and bounded, thus it converges. The resulting graph for `remaps` is shown in Figure 5.

Let us assume a $\mathcal{O}(1)$ set's element put, get and in-test implementation. Let m be the number of vertices in $\mathcal{G}_R$, p the number of distributed arrays, q the maximum

[3] As a convention in the interpretation of the remapping graph, remappings at vertex v for array $\mathtt{A} \in \mathrm{S}(v)$ will not be generated if $\mathrm{L}_{\mathtt{A}}(v) = \emptyset$.

number of different mappings for an array and r the maximum number of predecessors for a vertex. Then the worst case time complexity of the optimization, for a simple iterative implementation, is $\mathcal{O}(m^2pqr)$.

This optimization is correct and the result is optimal:

Theorem 1 *The computed remappings (from new reaching to remaining leaving) are those and only those that are needed (according to the static information provided by the data flow graph):* $\forall v$, $\forall \mathtt{A} \in S(v) \wedge Used_{\mathtt{A}}(v)$, $\forall a \in R_{\mathtt{A}}(v)$,
$\exists w$ *and a path from* w *to* v *in* $\mathcal{G}_R$*, so that* $a \in L_{\mathtt{A}}(w)$ *and* `A` *is not used on the path.*

Proof: construction of the path by induction on the solution of the data flow problem. Note that the path in $\mathcal{G}_R$ reflects an underlying path in the control flow graph with no use and no remapping of the array.

1.3 Discussion

- If subroutine local arrays are not used from the entry point in their initial mapping, the compiler may delay the allocation till a used mapping is needed, or chose another initial mapping among directly useful ones.

- Remappings involving unknown `X` mappings should be propagated to call sites in order to be instantiated.

- The set of needed remappings after this optimization may have been reduced or extended. What is minimized is the number of remappings performed at run-time, not those that must be addressed at compile time. Our compiler keeps a database of generated remapping codes in order not to generate some code twice.

- The remapping graph was presented at an intraprocedural level. It is natural to extend it to the interprocedural level, for instance by providing a summary of the entry and exit remappings to be used at the call sites foroptimizations. Remappings of arguments should be decided and performed at call site.

- $\mathcal{G}_R$ for `remaps` includes an edge from the entry to the exit vertex, because the `DO` loop may be empty and thus array `A` may reach the exit vertex without remapping. If the compiler can determine that the loop body is always executed, the skipping edge can be removed from the control graph, thus improving remapping graph $\mathcal{G}_R$ quality.

- In order to simplify the presentation, it was assumed that only one mapping for an array could leave a remapping statement. This is not necessarily the case in HPF. Thus several leaving mappings may be associated to a vertex and array, and for each of these mappings a set of reaching mappings. Care must also be taken in the building phase. Use-information must be attached to the leaving mappings.

- The runtime needs to keep track of the mapping status of each array to chose the right remapping routine when needed.

- Some additional benefits may be obtained by moving remapping in the control flow graph, in order to perform a remapping only when the array is to be actually referenced in its new shape.

References

[1] Ken Kennedy. A survey of data flow analysis techniques. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 5–54. Prentice-Hall, Inc., Engelwood Cliffs, 1979.

[2] Gary A. Kildall. A unified approach to global program optimization. In *Symposium on Principles of Programming Language*, pages 194–206, 1973.

[3] Hans Zima, Peter Brezany, Barbara Chapman, Piyush Mehrotra, and Andreas Schwald. Vienna Fortran - A Language Specification. ftp `cs.rice.edu` `public/HPFF/papers/vf.tex`, 1992. Version 1.1.

```
      SUBROUTINE remaps(A)
! distribute A... => A mapping: 0
      local arrays B, C
! template T, align B, C with T, distribute T... => B and C mappings: 0
      use C
      use B
1     redistribute T => B and C mappings: 1
      DO ...
2        remap A... => A mapping 1
         use A
3        remap A... => A mapping 2
         use A
      ENDDO
      use B
      END
```

Figure 1: Simple ADI-like structured example

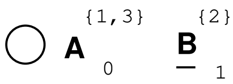

Figure 2: A simple vertex

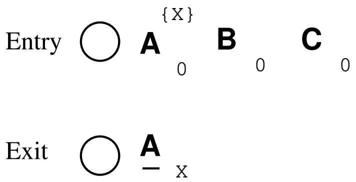

Figure 3: $\mathcal{G}_R$ for `remaps` at construction start

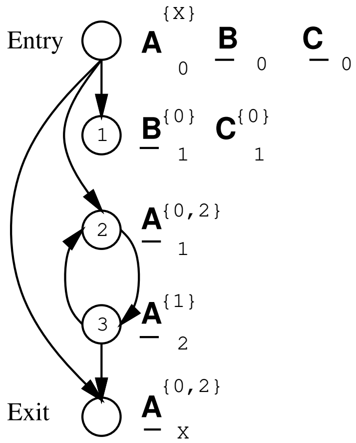

Figure 4: Initial $\mathcal{G}_R$ for `remaps`

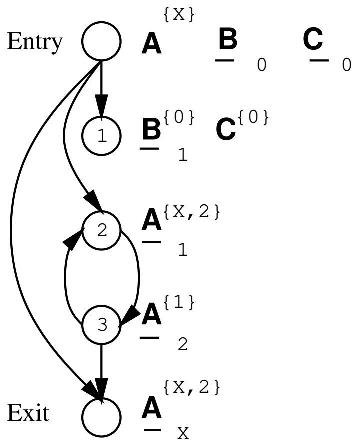

Figure 5: Optimized $\mathcal{G}_R$ for `remaps`