Chapter 7. Interprocedural Analysis Based on Guarded Array Regions

Zhiyuan Li¹, Junjie Gu², and Gyungho Lee³

¹ Department of Computer Science, University of Minnesota, 200 Union Street S.E., Minneapolis, MN 55455, USA
² Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303, USA
³ Dept. of Electrical and Computer Engineering, Iowa State University, Ames, IA 50011, USA

Summary. Array data flow information plays an important role for successful automatic parallelization of Fortran programs. This chapter proposes a powerful symbolic array data flow summary scheme to support array privatization and loop parallelization for programs with arbitrary control flow graphs and acyclic call graphs. Our approach summarizes array access information interprocedurally, using guarded array regions. The use of guards allows us to use the information in IF conditions to do path-sensitive data flow summary and thereby to handle difficult cases. We also provide a mechanism to overcome the disadvantages of non-closed union and difference operations. This improves not only the exactness of summaries, but also the efficiency of the summarizing procedure. Our preliminary result on array privatization shows that our summary scheme is fast and powerful.

Key words: Parallelizing compilers, array data flow analysis, interprocedural analysis, array privatization, guarded array regions, symbolic analysis.

1. Introduction

Interprocedural analysis, i.e. analyzing programs across routine boundaries, has been recognized as an essential part of program analysis. Parallelizing compilers, as a tool to enhance program efficiency on parallel computers, depend heavily upon this technique to be effective in practice.

Interprocedural analysis for scalars has been well studied. In contrast, interprocedural analysis for array references is still an open issue. Currently, there are two ways to extend program analysis across a call site. The first and simpler way is to use inline expansion. The drawbacks of inlining are twofold. First, even if the program size does not increase after inline expansion, the loop body containing routine calls may grow dramatically. This often results in much longer compile time and much larger consumption of memory resources [21] because many compiler algorithms dealing with loops have non-linear complexity with respect to the loop body’s size. Because a routine is analyzed each time it is inlined, duplicate analysis can also cause parallelizing compilers to be inefficient. Second, some routines are not amenable to inline expansion due to complicated array reshaping and must be analyzed without inlining.
These drawbacks of inlining make a summary scheme often a more desired alternative. Early works on summary schemes summarize the side effects of a called routine with sets of array elements that are modified or used by routine calls, called MOD sets and USE sets respectively. Data dependences involving routine calls can be tested by intersecting these sets. Existing approaches can be categorized according to methods of set representation. Convex regions [27, 28] and data access descriptors [2] define sets by a system of inequalities and equalities, while bounded regular sections [5, 15] use range tuples to represent sets. Even though bounded regular sections are less precise than convex regions and data access descriptors, they are much simpler and are easy to implement.

The commonalities of these previous methods are as follows. First, because they are path-insensitive, IF conditions are not taken into account when program branches are handled. Second, they are flow-insensitive, so only MOD/USE sets of array elements, which are modified or used respectively, are summarized. Third, they use a single descriptor (a single regular section, a single convex region, etc.) to summarize multiple references to the same array. Because union operations are not closed, approximations have to be made in order to represent union results in a single convex region or a single regular section. This causes the loss of summary information. Therefore, these methods are insufficient for optimization such as array privatization.

Recently, flow-sensitive summary, or array data flow summary, has been a focus in the parallelizing compiler area. The most essential information in array data flow summary is the upward exposed use (UE) set [7, 12, 14, 17, 29]. Our work [12] and that of M. Hall, et al. [14] use either a list of regular sections or a list of convex regions to summarize each array in order to obtain more precise information than that provided by a single regular section or convex region. Our work is unique in its use of guarded array regions (GAR’s), providing the conditions under which an array region is modified or upward-exposedly used. This is in contrast to path-insensitive summaries which do not distinguish summary sets for different program paths. Compared to other approaches [7] which handle more restricted cases of IF conditions, our approach seems to be more general.

In this chapter, we will describe our array data flow summary based on guarded array regions in the context of parallelizing compilers. The techniques involved should also be applicable to various programming supporting systems which deal with arrays, e.g. compilers of explicit parallel programs. The remainder of the chapter is divided into the following sections. In section 2, we present brief background information. Section 3 discusses guarded array regions, followed by our summary algorithm in section 4. We address the implementation issues in section 5. The preliminary results using GAR’s for array privatization are shown in section 6. Section 7 briefly discusses the related works. Finally, we conclude this chapter in section 8.