Efficient Dependence Analysis for Java Arrays

Vivek Sarkar and Stephen Fink

IBM Thomas J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598, USA

Abstract. This paper studies dependence analysis for Java arrays, emphasizing *efficient solutions* that avoid a large compile-time overhead. We present a new approach for dependence analysis based on sparse congruence partitioning representations in SSA form. Since arrays in Java are dynamically allocated, our approach takes pointer-induced aliasing of array objects into account in conjunction with analysis of index values. We present experimental results to evaluate the effectiveness of our approach, and outline directions for further improvements.

1 Introduction

Parallelizing and optimizing compilers perform array dependence analysis to aid parallelizing transformations and enhance back-end optimizations. The core problem addressed by dependence analysis ask whether and under what conditions two array references may interfere. The bulk of past work on array dependence analysis has focused on imperative programming languages such as Fortran and C, and has resulted in a wide range of data dependence tests based on *symbolic analysis of index values* (*e.g.,* see [14,2,8,10,9,11,3]). Java, with dynamically allocated arrays, also requires *pointer-induced alias analysis* (*e.g.,* see [5,13,12]) of array objects as part of array dependence analysis.

We examine an approach to dependence analysis for Java arrays, based on sparse congruence partitioning in SSA form and accounting for pointer-induced aliasing of array objects. The main features of our approach are:

1. *Congruence Partitioning* [1] based on SSA form provides an efficient approximate analysis of when two index values are "definitely same" (DS).
2. A new *Uniformly-Generated Partitioning* provides provides an efficient approximate analysis of when two index values are "uniformly generated" [7] *i.e.,* when the two index values differ by a compile-time constant.
3. The *Inequality Graph* representation from [3] provides an efficient approximate analysis of when two index values are related by a $<$ or $>$ inequality.

These representations all rely on an extended SSA form. Since the compiler uses only efficient *sparse* dataflow techniques, these techniques are ameneble for runtime or dynamic (JIT) compilation. Experimental results show that congruence partitioning techniques catch a substantial number of cases, but also indicate significant opportunities for improvement with better algorithms and inter-procedural analysis.

R. Sakellariou et al. (Eds.): Euro-Par 2001, LNCS 2150, pp. 273–277, 2001.

2 Background

The dependence analysis approach presented in this paper builds on the SSA congruence partitioning algorithm introduced in [1], and the Inequality Graph from [3].

We first partition SSA reference variables into equivalence classes. The result is represented as $\text{REF}_{\cong}$, where $\text{REF}_{\cong}(a)$ is the equivalence class for variable a. Given two SSA variables containing object references, a and b, if $\text{REF}_{\cong}(a) = \text{REF}_{\cong}(b)$ then the (unique) definitions of a and b are guaranteed to be "definitely same" *i.e.,* $\text{REF}_{\cong}.DS(a,b) = \text{true}$.

Further, as described in [6], if $\text{REF}_{\cong}(a) \neq \text{REF}_{\cong}(b)$, and both a and b belong to equivalence classes that included the result of a `new` operation, then a and b are guaranteed to be "definitely different" *i.e.,* $\text{REF}_{\cong}.DD(a,b) = \text{true}$. This also holds if a's equivalence class includes the result of a `new` operation, and b's equivalence class includes a parameter.

The *Inequality Graph* (IG) representation is used to provide an efficient approximate solution to the problem of determining when two index values are related by a $<$ or $>$ inequality. As described in [3], a demand-driven traversal of the inequality graph can establish that two index values belong to an inequality relation. We omit further details due to space constraints; see [3] for more details.

3 Uniformly-Generated Partitioning of Integer Variables

Past work has shown that the most common case in data dependence analysis compares index expressions that differ by a compile-time constant [8,10]. Such index expressions are said to be "uniformly-generated" [7]. In this section, we describe the *uniformly-generated partitioning* representation, which is as an extension to congruence partitioning and applies to all integer-like variables.

The partitioning is represented by three structures – $\text{INDEX}_{\cong}$, INDEX_OFFSET, and INDEX_REP, where $\text{INDEX}_{\cong}(a)$ is the "uniformly-generated" equivalence class for variable a. Given two SSA variables, a and b, if $\text{INDEX}_{\cong}(a) = \text{INDEX}_{\cong}(b)$ then the (unique) definitions of a and b are guaranteed to compute values that differ by a compile-time constant. The values of a and b are related by the identity, $a = b + \text{INDEX\_OFFSET}(a) - \text{INDEX\_OFFSET}(b)$, where $\text{INDEX\_OFFSET}(a)$ contains a constant offset that relates the value of a to the value of a hypothetical *representative variable*, $\text{INDEX\_REP}(a)$ for the equivalence class containing variable a by the identity, $a = \text{INDEX\_REP}(a) + \text{INDEX\_OFFSET}(a)$.

If $\text{INDEX}_{\cong}(a) = \text{INDEX}_{\cong}(b)$ and $\text{INDEX\_OFFSET}(a) = \text{INDEX\_OFFSET}(b)$, then the definitions of a and b are guaranteed to be "definitely same", indicated by $\text{INDEX}_{\cong}.DS(a,b) = \text{true}$. If $\text{INDEX}_{\cong}(a) = \text{INDEX}_{\cong}(b)$ and $\text{INDEX\_OFFSET}(a) \neq \text{INDEX\_OFFSET}(b)$, then the definitions of a and b are guaranteed to be "definitely different *i.e.,* $\text{INDEX}_{\cong}.DD(a,b) = \text{true}$.

The three structures are initialized as follows. First, congruence partitioning is performed on integer variables, and $\text{INDEX}_{\cong}$ is initialized to the resulting partition. A representative variable is arbitrarily selected from each equivalence

class in the partitioning. For each integer variable, a, INDEX_OFFSET(a) is set $= 0$, and INDEX_REP(a) is set to the representative variable for a's equivalence class.

Next, we merge congruence classes as follows. For each instruction of the form, $a := b + constant$, the equivalence classes containing a and b are merged. REF$_{\cong}$ is updated to reflect the result of the merge. INDEX_REP(a), the representative variable for a's class is arbitrarily chosen as the representative variable for the merged class. Further, the offset values for b and all variables that were in the same class as b are updated such that INDEX_OFFSET(b) := INDEX_OFFSET(a) − *constant*.

4 Algorithm

This section lays out the entire approach for intraprocedural dependence analysis considered in this paper. First we construct the sparse data structures as described; namely, SSA form, the congruence partitioning, and the uniformly-generating congruence partitions, and the inequality graph.

Next, we query these data structures to compare. a pair of one-dimensional array accesses, $a[i]$ and $b[j]$, in the extended SSA form. We perform the following steps in an attempt to determine whether these array accesses are "definitely-different" or "definiately-same". If none of the previous steps has a a conclusive answer, Step 6 returns *unknown* as a conservative default solution.

1. *Type propagation and disambiguation*
   if a and b cannot have overlapping types **then return** *definitely-different*;
2. *Definitely-different test for object references using* REF$_{\cong}$
   if REF$_{\cong}$.$DD(a, b) = true$ **then return** *definitely-different*;
3. *Definitely-different test for index values using* INDEX$_{\cong}$
   if INDEX$_{\cong}$.$DD(a, b) = true$ **then return** *definitely-different*;
4. *Definitely-same test using* REF$_{\cong}$ *and* INDEX$_{\cong}$
   if REF$_{\cong}$.$DS(a, b) = true$ **and** INDEX$_{\cong}$.$DS(a, b) = true$ **then return** *definitely-same*;
5. *Traversals of the Inequality Graph, IG*
   Traverse IG starting at i and attempt to prove $i < j$ or $j > i$.
6. *Otherwise*
   return *unknown*;

5 Experimental Results

We present results using the implementation of extended SSA form in the Jalapeño dynamic optimizing compiler [4,6]. We performed the dependence analysis test for each pair of distinct array references in each innermost loop of each method executed.

Table 1 presents "static" counts of array reference pairs examined. The `aload` and `astore` columns represent the total number of `aload` and `astore` instructions encountered in innermost loops. The `# pairs` column contains the number of pairs of distinct `aload`/`astore` instructions found in innermost loops. The remaining columns break down the number of pairs into six categories:

Table 1. Summary of dependence test results on pairs of array references in innermost loops. The table reports on the 7 SPECjvm98 codes, and on two parallel computational fluid dynamics codes described in [**?**]

Benchmark	aload	astore	# pairs	type.DD	ref.DD	index.DD	graph.DD	DS	unresolved
compress	20	15	13	0	0	0	0	1 (7.7%)	12 (92.3%)
jess	255	42	793	335 (42%)	0	3 (0.4%) 1	(0.1%)	32 (4.0%)	422 (53%)
db	44	13	37	0	0	0	0	7 (19%)	30 (81%)
javac	192	71	181	30 (17%)	1 (1.2%)	12 (6.6%)	0	22 (12%)	116 (64%)
mpegaudio	172	43	658	295 (45%)	0	103 (16%)	2 (0.3%)	26 (4.0%)	232 (35%)
mtrt	78	13	134	28 (21%)	3 (2.2%)	5 (3.7%)	0	6 (4.5%)	92 (69%)
jack	60	11	60	0	0	0	0	2 (3.3%)	58 (97%)
laura	122	54	343	127 (37%)	22 (6.4%)	105 (31%)	0	6 (1.7%)	83 (24%)
2dtag	24	22	247	123 (50%)	0	11 (4.5%)	0	4 (1.6%)	109 (44%)
section2	123	53	403	79	0	92	0	45	187

type.DD The number of pairs which type propagation determines to be *definitely-different* (Step 1).
ref.DD The number of pairs which simple alias analysis determines to be *definitely-different* (Step 2).
index.DD The number of pairs which uniformly-generated congruence partitions determine to be *definitely-different* (Step 3).
graph.DD The number of pairs which the inequality graph traversal determines to be *definitely-different* (Step 5).
DS The number of pairs which congruence partitioning determines both the index and array reference to be *definitely-same* (Step 4).
unknown All remaining pairs (Step 6).

The results show that Java's strong type system helps disambiguate many array references. The two variants of congruence partitioning are the next most effective. The simple alias analysis and inequality graph traversals are mostly ineffective. The totality of techniques appears to be most effective on the two numerical CFD codes (laura and 2dtag). The number of unresolved pairs ranges from 24% to 97%. However, as some pairs are neither DD nor DS, even exact analysis would have non-zero unresolved pairs. A limit on analysis precision remains open for future work.

These results suggest that there may be room for increased precision with more powerful techniques. It is not surprising that intra-procedural alias analysis provided little help for DD. We experimented with assuming that distinct parameters are not aliased, which determined 3 additional pairs to be DD. We are also investigating more powerful variants of the inequality graph and other sparse demand-driven dataflow techniques.

6 Conclusions and Future Work

This paper presented a new approach for dependence analysis based on sparse congruence partitioning representations built on SSA form. Our experimental results demonstrate some effectiveness for dependence tests on array references based on standard and uniformly generated congruence partitioning. However, there may be significant opportunities for improvement with interprocedural analysis and/or improved dataflow propagation of inequalities.

References

1. B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In ACM, editor, *POPL '88. Proceedings of the conference on Principles of programming languages, January 13–15, 1988, San Diego, CA*, pages 1–11, New York, NY, USA, 1988. ACM Press. 273, 274
2. U. Banerjee. *Loop transformations for restructuring compilers: the foundations.* Kluwer Academic Publishers, Boston, MA, 1993. 273
3. R. Bodik, R. Gupta, and V. Sarkar. ABCD: Eliminating Array Bounds Checks on Demand. In *SIGPLAN 2000 Conference on Programming Language Design and Implementation*, June 2000. 273, 274
4. M. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Dynamic Optimizing Compiler for Java. In *ACM Java Grande Conference*, June 1999. 275
5. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994. *SIGPLAN Notices, 29(6).* 273
6. S. Fink, K. Knobe, and V. Sarkar. Unified analysis of array and object references in strongly typed languages. In *Seventh International Static Analysis Symposium (2000)*, June 2000. 274, 275
7. K. Gallivan, W. Jalby, and D. Gannon. On the Problem of Optimizing Data Transfers for Complex Memory Systems. *Proceedings of the ACM 1988 International Conference on Supercomputing*, pages 238–253, July 1988. 273, 274
8. G. Goff, K. Kennedy, and C.-W. Tseng. Practical dependence testing. In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 15–29, 1991. *SIGPLAN Notices, 266.* 273, 274
9. M. R. Haghighat and C. D. P. Hronopoulos. Symbolic analysis for parallelizing compilers. *ACM Transactions on Programming Languages and Systems*, 18(4):477–518, July 1996. 273
10. D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 1–14, 1991. *SIGPLAN Notices, 266.* 273, 274
11. W. Pugh and D. Wonnacott. Eliminating false data dependences using the omega test. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation, San Francisco, California*, pages 140–151, June 1992. 273
12. B. Steensgaard. Points-to analysis in almost linear time. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 32–41, Jan. 1996. 273
13. R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995. *SIGPLAN Notices, 30(6).* 273
14. M. J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. In the series, Research Monographs in Parallel and Distributed Computing. 273