Heap Dependence Analysis for Sequential Programs

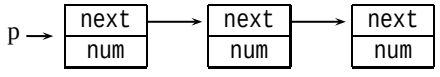
Barnali BASAK, Sandeep DASGUPTA and Amey KARKARE
{barnali, dsand, karkare}@cse.iitk.ac.in
Department of CSE, IIT Kanpur, India

Abstract. In this paper we demonstrate a novel intra-procedural technique for detecting heap dependences in sequential programs that use recursive data structures. The novelty of our technique lies in the way we compute, for each statement, abstract *heap access paths* that approximate the locations accessed by the statement, and the way we convert these paths into equations that can be solved using traditional tests, e.g. GCD test, Banerjee test and Lamport test. The dependence test also uses a field sensitive shape analysis to detect dependences among heap locations arising out of sharing within the data structure. In presence of loops, the technique can be used to discover *loop dependences*. i.e. the dependence among two different iterations of the same loop. This information can be used by a parallelizing compiler to transform sequential input program for better parallel execution.

Keywords. Dependence Analysis, Shape Analysis, Parallelizing Compilers, Recursive Data Structures

Introduction

In the recent arena of parallel architectures (multi-cores, GPUs, etc.), software side lags behind hardware in terms of parallelism. Parallelization of sequential programs, without violating their correctness, is a key step in increasing their performance and efficiency. Over the past years, lot of work has been done on automatically parallelizing sequential programs. These approaches have mainly been developed for programs written in languages, such as FORTRAN, having only static data structures (fixed sized arrays) [1,2,3,4]. Almost all programming languages today use the heap for dynamic memory structures. Therefore, any parallelization must also take into account the data dependency due to the access of common heap locations. Finding parallelism in sequential programs written in languages with dynamically allocated data structures, such as C, C++, JAVA, LISP etc., has been less successful. One of the reason being the presence of pointer-induced aliasing, which occurs when multiple pointer expressions refer to same storage location. Compared to the analysis of static and stack data, analyzing properties of heap data is challenging because the structure of heap is unknown at compile time, it is also potentially unbounded and the lifetime of a heap object is not limited by the scope that creates it. As a consequence, properties of heap (including dependence) are approximated very conservatively. The approximation of the heap dependence information inhibits the parallelization. The following example motivates the need for a precise dependence analysis.



num is the data field.
next is the pointer to the next element.

(a) A singly linked list

```
      ...
S1.    p = list;
       while (p→next != NULL) {
S2.      q = p→next;
S3.      ...= q→num;
S4.      r = q→next;
S5.      r→num = ...;
S6.      p = r;
       }
       ...
```

(b) Code traversing the list

Figure 1. A motivating example

Example 1 Figure 1 shows a singly linked list and a code fragment traversing that list. The performance of the code can be improved if the loop can be executed in parallel. However, without the knowledge of precise heap dependences, we have to assume worst case scenario, i.e., the location read by the statement S3 in some iteration could be the same as the location written by the statement S5 in some other iteration. In that case, it is not possible to parallelize the loop.

Our dependence analysis can show that the locations read by S3 and those written by S5 are mutually exclusive. Further, it also shows the absence of any other dependences. This information, along with the information from classical control and data dependence analysis, can be used by a parallelizing compiler to parallelize the loop. □

The rest of this paper explains our approach for a practical intra-procedural heap data dependence analysis. As it is understood that we are only talking about data dependences, we drop the term data in the rest of the paper. We first describe a shape analysis that is used by our analysis to detect sharing (also called interference) among the data structures created on the heap. The details of dependence analysis are explained next. We then present our method to handle loops in a more precise way. We finish the paper by giving the directions for the future research.

1. Shape Analysis

The goal of our shape analyzer is to detect the shape of the data structure pointed to by the heap directed pointers at each program point. Our approach is similar to the work proposed by Ghiya et. al. [5] in that it also uses the *Direction Matrix* and the *Interference Matrix* to keep track of shapes of data structures. However, our shape analysis is *field sensitive*; it remembers abstracted paths between two heap nodes. The path abstraction is done by using fixed length prefixes (sequence of field names) of the paths between two heap nodes. As the number of paths starting with the same fixed length prefixes may be unbounded so we use k-limiting on that number i.e. only k such paths will be considered.

The novelty of our approach lies in the way we use field information to remember the paths that result in a particular shape (Tree, DAG, Cycle). We associate the field information with a shape in two ways: (a) through boolean functions that capture the shape transition due to change in a particular field, and (b) through matrices that store

the field sensitive path information among two pointer variables. This allows us to easily identify transitions from Cycle to DAG, from Cycle to Tree and from DAG to Tree, thus making the shape more precise. This is an improvement over earlier approaches like Ghiya et. al. [5] where once a data structure is marked cyclic, it remains so for the rest of the analysis.

For dependence detection, our shape analysis technique provides an interface function `isInterfering(`p`,` α`,` q`,` β`)`. For heap pointers p, q and field sequences $\alpha, \beta, \alpha', \beta'$, this function returns true if the paths $p.\alpha'$ and $q.\beta'$ interfere (potentially reach the same heap node at run-time), and α' and β' are prefixes of α and β respectively. The result of the interface function is based on the following observations:

1. If the shape attribute of a pointer variable is Tree, then two access paths rooted at that variable cannot interfere. Two access paths can only visit a common node if the paths are equivalent. Let `lp` be the pointer variable. Hence `lp`$\rightarrow$`f` and `lp`$\rightarrow$`f` are equivalent paths leading to a common node, whereas `lp`$\rightarrow$`f` and `lp`$\rightarrow$`g` lead to different nodes.
2. If the shape attribute is DAG and if it is traversed using a sequence of fields, then every sub-sequence accesses a distinct node. If an access path is a proper sub-path of another access path then they surely visit distinct nodes. However, if the paths equivalent or distinct (i.e. having different pointer field references), they may access a common node. For example, `lp`$\rightarrow$`f` is a proper sub-path of `lp`$\rightarrow$`f`$\rightarrow$`g` whereas, `lp`$\rightarrow$`f` and `lp`$\rightarrow$`g` are not. Hence in the former case they do not share a common node, whereas in later case they might result in sharing of node.
3. If shape attribute of a pointer variable is Cycle, we make conservative decision such that the access paths originating from that pointer interfere.

The details of the path abstraction and the shape analysis can be found elsewhere [6].

2. Dependence Analysis

Two statements are said to be heap dependent on each other if both statements access the same heap location and at least one of the statements writes to that location. We have developed a novel technique which finds out heap induced dependencies, between any two statements in the program. The novelty of our approach lies in the separation of shape analysis phase from the dependence detection phase For each statement our analysis computes two sets of heap access paths (a) *Read set*: the set of paths which are accessed to read a heap location, and (b) *Write set*: the set of paths which are accessed to write a heap location.

Our approach is conservative in the sense that the read set and write set we compute for a statement are over approximations of the actual locations that are read or written by the statement. Therefore it is possible that our analysis reports two statement to be dependent when they are not really dependent on each other. However, this can inhibit some parallelizing optimization but can not result in an incorrect parallelization. Function calls are handled by using conservative read/write sets that over approximate the heap locations that could potentially be read or written inside the called function.

```
analyze(f, k) {
  initialize(); /* Initialize all parameters and globals  */
  States = computeStates(k)
  computeReadWrite(States)
}
```

Figure 2. Algorithm to analyze a function f for dependence detection. Parameter k is used for limiting the length of access paths, to keep the analysis bounded.

Figure 2 gives top level pseudo code for analyzing a function f in the program. The pseudo codes for the utility functions used by the above code is given in [7]. We describe their functionality in brief.

Function initialize() initializes the parameters of the function and global variables with symbolic values. The initialization information is also accessible to the shape analyzer that requires it for making interference decisions.

The function computeStates(k) computes the bindings of pointer variables to the access paths using traditional iterative data flow analysis. The access paths are either symbolic locations, e.g. l_0, or symbolic location followed by pointer fields, e.g. $l_0 \rightarrow$next$\rightarrow$next. To guarantee termination, we limit the length of access paths to k, that is a parameter to computeStates. A special summary field '$*$' is used to limit the access paths. which stands for any field dereferenced beyond length k. Hence, for k = 1, all the access paths in the set {$l_0\rightarrow$next$\rightarrow$next, $l_0\rightarrow$next$\rightarrow$next$\rightarrow$next, $l_0\rightarrow$next$\rightarrow$next$\rightarrow$next$\cdots\rightarrow$next} can be abstracted as a single summarized path $l_0\rightarrow$next$\rightarrow *$. Similarly, assuming a data structure has two reference fields left and right, the summarized path $l_0 \rightarrow$ left $\rightarrow$ right $\rightarrow *$ could stand for any of the access paths $l_0 \rightarrow$ left $\rightarrow$ right $\rightarrow$ left, $l_0 \rightarrow$ left $\rightarrow$ right $\rightarrow$ right, $l_0 \rightarrow$ left $\rightarrow$ right $\rightarrow$ left $\rightarrow$ left, $l_0 \rightarrow$ left $\rightarrow$ right $\rightarrow$ left $\rightarrow$ right and more such paths.

Given a variable to access path mapping States, computeReadWrite(States) computes read and write sets for each statement in the program in a single pass. This information can be used for computation of various dependencies (flow, anti or output). Let S and S' be statements in the program such that there exists an execution path from S to S'. Then, the dependence of S' on S is computed as follows:

$$\begin{aligned}
\mathsf{interfere}(\text{set}_1,\ \text{set}_2) &\equiv \text{isInterfering}(p,\alpha,q,\beta) \text{ where } p.\alpha \in \text{set}_1 \wedge q.\beta \in \text{set}_2\\
\text{flow-dep}(S,S') &\equiv \mathsf{interfere}(\text{write}(S),\ \text{read}(S'))\\
\text{anti-dep}(S,S') &\equiv \mathsf{interfere}(\text{read}(S),\ \text{write}(S'))\\
\text{output-dep}(S,S') &\equiv \mathsf{interfere}(\text{write}(S),\ \text{write}(S'))
\end{aligned}$$

where isInterfering is the function provided by shape analysis.

Example 2 Table 1 shows the state (pointer variables and symbolic memory locations referred by the variables) and the read and write sets for each statement in the example code of Figure 1. From the table, we can infer the following dependences:

1. loop independent anti-dependence from statement S3 to statement S5
2. loop carried flow-dependence from statement S5 to statement S3

Note that the dependences inferred by our analysis are a super-set of actual dependences that exist in the program. □

Table 1. Simple dependence analysis for code in Figure 1

Stmt	Variables and Locations of Interest	Read Set	Write Set
S1	$p \equiv \{l_0\}$	$\emptyset$	$\emptyset$
S2	$q \equiv \{l_0 \rightarrow \texttt{next}, l_0 \rightarrow \texttt{next} \rightarrow *\}$	$\emptyset$	$\emptyset$
S3	$q \equiv \{l_0 \rightarrow \texttt{next}, l_0 \rightarrow \texttt{next} \rightarrow *\}$	$\{l_0 \rightarrow \texttt{next}, l_0 \rightarrow \texttt{next} \rightarrow *\}$	$\emptyset$
S4	$r \equiv \{l_0 \rightarrow \texttt{next} \rightarrow *\}$	$\emptyset$	$\emptyset$
S5	$r \equiv \{l_0 \rightarrow \texttt{next} \rightarrow *\}$	$\emptyset$	$\{l_0 \rightarrow \texttt{next} \rightarrow *\}$
S6	$p \equiv \{l_0 \rightarrow \texttt{next} \rightarrow *\}$	$\emptyset$	$\emptyset$

l_0 is the symbolic location representing the value of the variable `list` at the start of the program.

Next we explain how we can further refine our dependence analysis to filter out some spurious dependences.

3. Loop Dependence Analysis

As can be seen from the Example 2 above, our approach, as explained earlier, does not work well for loops. This is because it combines the paths being accessed in different iterations of a loop. To get better result in presence of loops we need to keep the accesses made by different iterations of a loop separate. To do so, we have devised another novel approach, which works as follows: Given a loop, we first identify the navigators[1] [8] for the loop, then by a single symbolic traversal over the loop, we compute the read and write accesses made by each statement in terms of the values of the navigators. Using the iteration number as a parameter, the read and write sets are generalized to *symbolic access paths* [9] to represent arbitrary iteration of the loop.

Let S and S' be two statements inside a loop. Further, let write(S,i) denote the set of access paths written by statement S in the iteration number i, and let read(S',j) denote the set of access paths read by statement S' in the iteration number j. Then,

- S' is loop independent flow dependent on S if there is an execution path from S to S' that does not cross the loop boundary and there exist i within loop bounds[2] such that interfere(write(S,i), read(S',i)) is true.
- S' is loop dependent flow dependent on S if there exist i and j within loop bounds such that $j > i$, and interfere(write(S,i), read(S',j)) is true.

We can similarly define loop independent and loop carried anti-dependence and flow-dependence. The following example explains our approach for the loop dependence detection.

Example 3 Consider Figure 3. Assume in each case l_0 is the symbolic variable pointed to by the variable `list`.

For the code in Figure 3(a), the navigator is $\langle l_0, \texttt{next} \rightarrow \texttt{next} \rangle$. Using i to represent the iteration number, the generalized access path read by S11 is $l_0 \rightarrow \texttt{next}^{2i} \rightarrow \texttt{next}$ and the generalized access path written by S12 is $l_0 \rightarrow \texttt{next}^{2i+1} \rightarrow \texttt{next}$ Clearly there is no loop independent dependence. To find out loop carried dependence, we have to find out

[1] a navigator consists of a pointer variable and a set of field references that are used to traverse a data structure inside the loop.

[2] in case loop bounds can not be computed at compile time, we can assume them to be $(-\infty, \infty)$

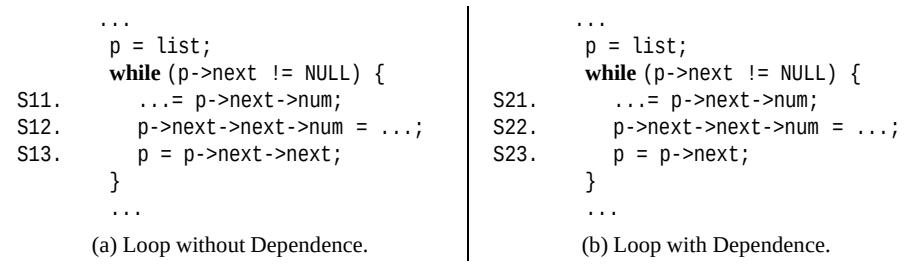


Figure 3. Identifying Loop Dependences

whether for iterations i and j, the two paths point to the same heap location. This reduces to finding out if there is a possible solution to the following equation:

$$l_0 \rightarrow \mathtt{next}^{2i} \rightarrow \mathtt{next} = l_0 \rightarrow \mathtt{next}^{2j+1} \rightarrow \mathtt{next}$$

In other words, we have to find out if the following equation has integer solutions:

$$2 * i = 2 * j + 1$$

GCD [4] or Lamport [10] test tell us that this equation can not have integer solutions. Thus, there is no dependence among the statements.

For the code in Figure 3(b), the navigator is $\langle l_0, \mathtt{next} \rangle$. In this case the equation to find out the loop carried dependences among statements S21 and S22 reduces to:

$$i = j + 1$$

which has integer solutions. So we have to conservatively report dependence between the two statements

In both the cases, we also need the shape analyzer to assert that there is no sharing within the underlying data structure. Had there been a sharing, we would have to report conservatively that there exists a dependence.

4. Related Work

Data dependence analysis for sequential programs, working on only static and stack related data structure, such as array, is well explored in literature [1,3,2,4] etc. Our work extends the work to handle heap data structure. Various approaches have been suggested for data flow analysis of programs in the presence of dynamic data structures. We describe briefly some of the earlier work done in this area.

The work by Hendren et al. in [11] considers shape information and approximates the relationships between accessible nodes in larger aggregate data structures. These relationships are represented by path expressions, a restricted form of regular expressions, and are encoded in path matrices. Such matrices are used to deduce the interference information between any two heap nodes, and to extract parallelism. Their method focuses on three levels of parallelization; (a) if two statements can be executed in parallel, (b) identifies procedure-call parallelism, and (c) whether two sequences of statements can be parallelized.

Ghiya et. al. [8] uses coarse characterization of the underlying data structure as Tree, DAG or Cycle. They compute complete access paths for each statement in terms of *anchor* pointer, which points to a fixed heap node in the data structure within the whole body of the program. The test for aliasing of the access paths, relies on connection and shape information that is automatically computed. They have also extended their work to identify loop carried dependences for loop level parallelism.

Hwang and Saltz [12] present a technique to identify parallelism in programs with cyclic graphs. The method identifies the patterns of the traversal of program code over the underlying data structure. In the next step the shape of the traversal pattern is detected. If the traversal pattern is acyclic, dependence analysis is performed to extract parallelism from the program.

Navarro et. al. in [13,14] propose a intra-procedural dependence test which intermixes shape analysis and dependence analysis together. During the analysis, the abstract structure of the dynamically allocated data is computed and is also tagged with read/write tags to find out dependencies. The resulting analysis is very precise, but it is costly. Further their shape analysis component is tightly integrated within the dependence analysis, while in our approach we keep the two separate as it gives us modularity and the scope to improve the precision of our dependence analysis by using a more precise shape analysis, if available. They have extended their dependence related work in [15], where they have implemented a context-sensitive interprocedural analysis which successfully detects dependences for both non-recursive and recursive functions.

Work done by Marron et. al. in [16] tracks a two program location, one read and one write location, for each heap object field. The technique uses an explicit store heap model which captures the tag information of objects for each program statement. The read and write information are used to detect dependences. This space effective and time efficient technique analyses bigger benchmarks in shorter time. But the effectiveness of this approach lies in the use of predefined semantics for library functions [17], which recognizes a traversal over a generic structure.

Our approach is closest to the technique proposed by Horwitz et. al. [18]. They also associate read and write sets with each program statement to detect heap dependencies. They have also proposed technique to compute dependence distances for loop constructs. However, there technique requires iterating over a loop till a fixed point is reached, which is different from our method of computing loop dependences as a set of equations in a single pass, and then solving these equations using classical tests.

5. Conclusion and Future Work

In this report we have presented our work on heap dependence analysis that can be utilized by a parallelizing compiler to parallelize sequential programs. Our method is divided into three phases - the shape analysis phase, the state computation phase, and the loop analysis phase, with carefully chosen interfaces between phases to combine work done by individual phases. This gives us the flexibility to work on testing and improving each phase independently. Our loop dependence analysis abstracts the dependence information in forms of linear equations, that can be solved using traditional dependence analysis tests [1,3,2] that exist for finding array dependences.

Our analysis is intra-procedural, and we use conservative approximation of function calls assuming worst case scenario. Our next challenge is to develop an inter pro-

cedural analysis to handle function calls more precisely. We have to further develop our shape analysis technique and the loop analysis to handle more of frequently occurring programming patterns to find precise dependences for these patterns. We also want to improve our summarization technique. Earlier we have used graph based approximations of access paths [19] to compute liveness of heap data. We plan to explore if the same summarization technique can also be applied here.

Finally, to show that our analysis is practical, we are developing a prototype model using GCC compiler framework to show the effectiveness on large benchmarks. However, this work is still in very early stages.

References

[1] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. *ACM Trans. Program. Lang. Syst.*, 9, 1987.

[2] Utpal Banerjee, Rudolf Eigenmann, Alexandru Nicolau, and David A. Padua. Automatic program parallelization, 1993.

[3] M. E. Wolf and M. S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, 2, 1991.

[4] Ken Kennedy and John R. Allen. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[5] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, New York, NY, USA, 1996. ACM.

[6] Sandeep Dasgupta. Precise shape analysis using field sensitivity. Master's thesis, IIT Kanpur, 2011. `http://www.cse.iitk.ac.in/users/karkare/MTP/2010-11/sandeep2010precise.pdf`.

[7] Barnali Basak. Heap dependence analysis for sequential programs. Master's thesis, IIT Kanpur, 2011. `http://www.cse.iitk.ac.in/users/karkare/MTP/2010-11/barnali2010heap.pdf`.

[8] Rakesh Ghiya, Laurie Hendren, and Yingchun Zhu. Detecting parallelism in C programs with recursive data structures. In *Compiler Construction*. 1998.

[9] Alain Deutsch. Interprocedural may-alias analysis for pointers: beyond k-limiting. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, New York, NY, USA, 1994.

[10] Leslie Lamport. The parallel execution of DO loops. *Commun. ACM*, 17, 1974.

[11] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. Parallel Distrib. Syst.*, 1, 1990.

[12] Yuan-Shin Hwang and Joel H. Saltz. Identifying parallelism in programs with cyclic graphs. *J. Parallel Distrib. Comput.*, 63, 2003.

[13] A. Navarro, F. Corbera, R. Asenjo, A. Tineo, O. Plata, and E. Zapata. A new dependence test based on shape analysis for pointer-based codes. In *Lang. and Comp. for High Performance Computing*. 2005.

[14] A. Tineo, F. Corbera, A. Navarro, and E. L. Zapata. A novel approach for detecting heap-based loop-carried dependences. In *Proceedings of the 2005 International Conference on Parallel Processing*, Washington, DC, USA, 2005. IEEE Computer Society.

[15] R. Asenjo, R. Castillo, F. Corbera, A. Navarro, A. Tineo, and E. Zapata. Parallelizing irregular C codes assisted by interprocedural shape analysis. In *2nd IEEE International Parallel & Distributed Processing Symposium*, 2008.

[16] M. Marron, D. Stefanovic, D. Kapur, and M. Hermenegildo. Identification of heap-carried data dependence via explicit store heap models. In *Languages and Compilers for Parallel Computing*, 2008.

[17] M. Marron, D. Kapur, D. Stefanovic, and M. Hermenegildo. A static heap analysis for shape and connectivity. In *Languages and Compilers for Parallel Computing*, 2006.

[18] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*, PLDI '89, New York, NY, USA, 1989. ACM.

[19] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. Heap reference analysis using access graphs. *ACM Transactions on Programming Languages and Systems*, 30(1), 2007.