Exploiting Locality and Parallelism in Pointer-based Programs

A. Navarro, F. Corbera, R. Asenjo, E. Gutiérrez, R.G. Valderrama, O. Plata and E.L. Zapata*
Dept. Computer Architecture, University of Málaga, Spain
{angeles,corbera,asenjo,eladio,rafagv,oscar,ezapata}@ac.uma.es

Abstract

While powerful optimization techniques are currently available for limited automatic compilation domains, such as dense array-based scientific and engineering numerical codes, a similar level of success has eluded general-purpose programs, specially symbolic and pointer-based codes. Current compilers are not able to successfully deal with parallelism in those codes. Based on our previously developed shape analysis techniques, we have designed preliminary methods to tackle the parallelism detection in those types of codes. As with parallelism, contemporary compilers cannot either successfully exploit locality exhibited in pointer-based programs. The locality problem comprises several aspects. In this paper we address two of the main aspects: data locality in cache hierarchy, and hiding of the processor-memory latency gap.

1 Introduction

Optimizing and parallelizing compilers rely upon accurate static disambiguation of memory references, i.e. determining at compiling time if two given memory references always access disjoint memory locations. Unfortunately the presence of alias in pointer-based codes makes memory disambiguation a non-trivial issue. An alias arises in a program when there are two or more distinct ways to refer to the same memory location. Program constructs that introduce aliases are arrays, pointers and pointer-based dynamic data structures.

*This research was supported by the Ministry of Education and Science (CICYT) of Spain under project TIC2003-06623

Over the past twenty years powerful data dependence analysis have been developed to resolve the problem of array aliases. The problem of calculating pointer-induced aliases, called pointer analysis, has also received significant attention over the past few years [1], [2], [3]. Pointer analysis can be divided into two distinct subproblems: stack-directed analysis and heap-directed analysis. We focus our research in the later, which deals with objects dynamically allocated in the heap. An important body of work has been conducted lately on this kind of analysis. A promising approach to deal with dynamically allocated structures consists in explicitly abstracting the dynamic store in the form of a bounded graph. In other words, the heap is represented as a storage shape graph and the analysis tries to estimate some shape properties of the heap data structures. This type of analysis is called *shape analysis* and our research group has developed a powerful shape analysis framework [4] .

Basically, the goal of this paper is to use our shape analysis framework to develop advanced compiling techniques for parallelism detection and exploiting locality in programs that operate with pointer-based data structures.

The rest of the paper is organized as follows: Section II briefly describes the key ideas under our shape analysis framework. With this background, in Section III we present our compiler techniques to automatically identify the parallel loops in codes based on dynamic data structures. On the other hand, in Section IV we focus in our preliminary work in order to exploit locality with the support of our shape analysis. Finally, in Section V we conclude with the main contributions and future work.

2 Shape Analysis Framework

Basically, our method is based on approximating by graphs all possible memory configurations that can appear after the execution of a statement in the code. We call a collection of dynamic structures a *memory configuration*. These structures comprise several memory chunks, that we call *memory locations*, which are linked by references. Inside these memory locations there is room for data and for pointers to other memory locations. These pointers are called *selectors*.

Note that due to the control flow of the program, a statement could be reached by following several paths in the control flow. Each "control path" has an associated memory configuration which is modified by each statement in the path. Therefore, a single statement in the code modifies all the memory configurations associated with all the control paths reaching this statement. Each memory configuration is approximated by a graph we call *Reference Shape Graph* (RSG). So, taking all this into account, we conclude that each statement in the code will have a set of RSGs associated with it.

2.1 RSGs and node properties

The RSGs are graphs in which nodes represent memory locations which have similar reference patterns. To determine whether or not two memory locations should be represented by a single node, each one is annotated with a set of properties. Now, if several memory locations share the same properties, then all of them will be represented by the same node. This way, a possibly unlimited memory configuration can be represented by a limited size RSG, because the number of different nodes is limited by the number of properties of each node. These properties are related to the "reference pattern" used to access the memory locations represented by the node. Hence the name *Reference Shape Graph*. These properties are described in [4], but two of them are summarized here because they are necessary in the following sections:

Share Information: This property can tell whether at least one of the locations represented by a node is referenced more than once from other memory locations. We use two kinds of attributes for each node: *SHARED(n)* states if any of the locations represented by the node n can be referenced by other locations by different selectors, and *SHSEL(n, sel)* points out if any of the locations represented by n can be referenced more than once by following the same selector sel from other locations.

Touch Information: This property is taken into account only inside loop bodies to avoid the summarization of already visited locations with non-visited ones. The touch information will be also the key tool in order to automatically annotate the nodes of the data structure which are written and/or read by the pointer statements inside loops.

As we have said, all possible memory configurations which may arise after the execution of a statement are approximated by a set of RSGs. We call this set *Reduced Set of Reference Shape Graphs* (RSRSG), since not all the different RSGs arising in each statement will be kept. On the contrary, several RSGs related to different memory configurations will be fused when they represent memory locations with similar reference patterns.

2.2 Generating the RSRSGs

To move from the "memory domain" to the "graph domain", the calculation of the RSRSGs associated with a statement is carried out by the **symbolic execution** of the program over the graphs. In this way, each program statement transforms the graphs to reflect the changes in memory configurations derived from statement execution. The **abstract semantic** of each statement states how the analysis of this statement must transform the graphs.

Let us illustrate all this with an example. In Figure 1 we can see a simple code with seven pointer statements. Our analyzer symbolically executes each statement to build the RSRSG associated with them. Actually, after the execution of the third statement we obtain an RSRSG with a single RSG which represents three different memory locations by three nodes; all of them of the same type, with

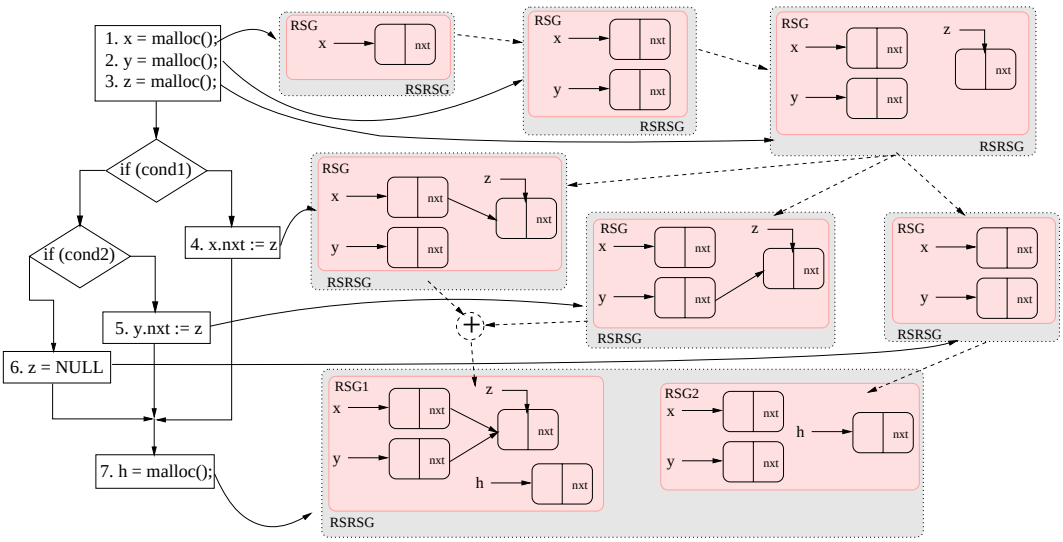

Figure 1: Building an RSRSG for each statement of an example code

the same *nxt* selector, but pointed to by different pointer variables (pvars). Now, this RSRSG is modified in three different ways because there are three different paths in the control flow graph, each one with a different pointer statement. All these paths join in statement 7, and after the execution of this statement we obtain an RSRSG with two RSGs. This is because the RSGs coming from statements 4 and 5 are compatible and can be summarized into a single one.

The whole symbolic execution process can be seen by looking at Fig. 2. For each statement in the code we have an input $RSRSG_i$ and the corresponding output $RSRSG_o$ representing the memory configurations after statement execution. During the symbolic execution of the statement all the rsg_{ij} belonging to $RSRSG_i$ are going to be updated. The first step comprises graph division to better focus on the several memory configurations represented by the RSG. Pruning removes redundant or nonexistent nodes or links that may appear after the division operation. Then the abstract interpretation of the statement takes place and usually the complexity of the RSGs grows. In order to counter this effect, the analysis carries out a compression operation. In this phase each RSG is simplified by the summarization of compatible nodes, to obtain the rsg^*_{ijk} graphs. Furthermore, some of the rsg^*_{ijk} can be fused into a single rsg_{ok} if they represent similar memory configurations. This operation greatly reduces the number of RSGs in the resulting RSRSG.

The abstract interpretation is carried out iteratively for each statement until we reach a fixed point in which the resulting rsg_{ok}'s associated with the statement does not change any more. This way, for each statement that modifies dynamic structures, we have defined the abstract semantics which describe how these statements modify the rsg_{ij}. We consider six simple instructions that deal with pointers:

```
x = NULL; x = malloc; x = y;
x->sel = NULL; x->sel = y; x = y->sel;
```

More complex pointer instructions can be built upon these simple ones and temporal variables. Due to space constraints we cannot formally describe the abstract semantics of each one of these statements (this can be found in [4]).

3 Parallelism Detection

We focus on detecting parallelism on loops that traverse heap-based recursive data structures. In general, for finding loop parallelism we need to detect the presence of *loop-carried dependences* (henceforth referred as LCDs). Two statements

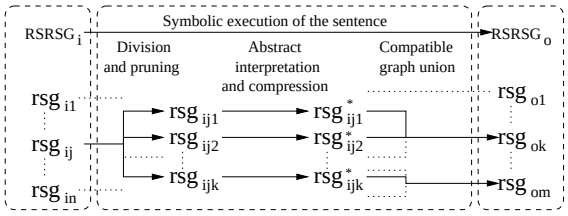

Figure 2: Schematic description of the symbolic execution of a statement.

```
while (p->next != NULL)
{
  S1:  tmp = p->i;
  S2:  p->next->i = tmp;
  S3:  p = p->next;
}
```

Figure 3: Example of a LCD between S1 and S2.

in a loop have a LCD, if a memory location accessed by one statement in a given iteration, is accessed by the other statement in a future iteration, with one of the accesses being a write accesses. The following loop in Figure 3, illustrates a LCD. This loop traverses a linked list and the reference `p->next->i` in the current iteration writes to the memory location, read by the reference `p->i` in the next iteration, thereby inducing a LCD between statements S1 and S2. The presence of LCDs in a loop indicates that its iterations are not independent, and hence cannot be executed in parallel.

A dependence analysis assisted by pointer analysis or shape analysis is key to identify the presence of LCDs in programs with pointers and dynamic recursive data structures. Next, we summarize some of the most relevant works in this research area.

3.1 Related work

Some of previous works combine dependence analysis techniques with pointer analysis [5, 6, 7, 8, 9, 10]. Horwitz et al. [6] developed an algorithm to determine dependence by detecting interferences in reaching stores. Larus and Hilfinger [7] propose to identify access conflicts on alias graphs using path expressions to name locations. Hendren and Nicolau [5] use path matrices to record connection information among pointers and present a technique to recognize interferences between computations for programs with acyclic structures. The focus of these techniques is on identifying parallelism at the function-call level and they do not consider the detection of loop parallelism, which is the focus in our approach.

More recently, some authors [9, 10] have proposed dependence analysis tests based on shape analysis in the context of loops that traverse dynamic recursive data structures, and these approaches are more related to our work. The mentioned authors base their shape analysis framework in a storeless abstraction that is computed for each program point. Their shape analysis consider three possible shape attributes: Tree, DAG or Cycle [9]. These authors have observed that knowledge about the shape of the data structure accessible from a heap-directed pointer, provides critical information for disambiguating heap accesses originating from it. The techniques based on this kind of shape information may be useful to provide that there are not data dependences between iterations and that the loop can be executed in parallel. However, the scope of applicability is limited to some particular cases (basically they are tree-oriented techniques). Let's see these approaches in more detail.

Ghiya and Hendren [9] propose a test for identifying LCDs that relies on the shape of the data structure being traversed, as well as on the detection of the *navigator* of the loop (the pointer used to traverse the data structure). Their approach used to identify LCDs follows the next steps:

(1) Once a navigator is identified, the access paths for the pointers in the statements being analyzed is constructed. These access paths are constructed with respect to the navigator.

(2) Using the shape information of the data structure and the computed access paths, the authors propose the following test to detect LCDs: (a) If the shape attribute is Tree, a dependence is reported if any of the access paths uses a navigator field as a pointer field (or selector). Otherwise, the analyzed statements with the Tree attribute does not lead to any dependence; (b) else, if the shape attribute is DAG or Cycle and the loop has been asserted by the programmer as acyclic, then a dependence is signaled when any of the access paths involves one or

more pointer fields. Otherwise, the analyzed statements in the asserted loop are not responsible of any LCD; (c) in any other case, the test return dependence.

This technique identifies parallelism in programs with tree-like data structure or loops that traverse DAG or Cycle structures and have been asserted by the programmer as acyclic. Note that the manual assertion of loops traversing cyclic data structures is a must in order to enable any automatic detection of parallelism. Another limitation is that data structures must remain static during the data traversal inside the analyzed loops.

In order to solve some of the previous limitations, Hwang and Saltz propose a new technique to identify LCDs in programs that traverse cyclic data structures [10]. This approach automatically identifies acyclic traversal patterns even in cyclic (Cycle) structures. For this purpose, the compilation algorithm isolates the traversal patterns from the overall data structure, and next, it deduces the shape of these traversal patterns. The novelty is in that they present a technique that perform traversal-pattern sensitive shape analysis. Once they have extracted the traversal-pattern shape information, dependence analysis is applied to detect parallelism.

In more detail, their approach to identify LCDs from programs that traverse dynamic data structures can be divided into the following steps:

(1) Traversal patterns inside loops are identified. The statements that construct the links of that traversal patterns are identified. Definition-use chains of the statements that define the traverse links in the data structure are constructed.

(2) Shape analysis is performed on the previously identified definition statements. This shape analysis gives the possible shapes of the traversal patterns. The authors consider only three possible shapes: Tree, DAG and Cycle.

(3) The shape information of the traversal patterns (but not of the global data structure) is used to determine if access conflicts occur between the sets of read and write references and facilitate the dependence analysis.

This technique identifies parallelism in programs that navigate cyclic data structures in a “clean” tree-like traverse. However the analysis can overestimate the shape of the traverse when the data structure is modified along the traverse, and in these situations, the shape algorithm detect DAG or Cycle traversal patterns, in which case dependence is reported.

3.2 Our approach

As we see, the previous data dependence tests based on shape analysis use as shape information a coarse characterization of the data structured being traversed (Tree, DAG, Cycle). One advantage of this type of analysis is that enables faster data flow merge operations and reduces the storage requirements for the analysis. However, it also means a loss of accuracy that prevent the parallelization of loops, specially when the data structure being visited is not a ”clean” tree or may contain cycles. Our approach, on the contrary, is based on a shape analysis that maintains topological information of the connections among the different nodes (memory locations) in the data structure. In fact, our representation of the data structure provides us a more accurate description of the memory locations reached when a statement is executed inside a loop.

Moreover, as we have seen in section 2.2, our shape analysis is based on the symbolic execution of the program statements over the graphs that represent the data structure at each program point. In other words, our approach does not extract the traversal paths (or navigator paths) and relies on a generic characterization of the data structure shape in order to prove the presence of LCDs. As we will see in this section, the novelty is that our approach symbolically executes the statements of the loop being analyzed, and let us to annotate the real memory locations reached by that statement with read/write information. This information will be used in order to find LCDs in a very accurate dependence test.

Our approach for finding parallelism focus in loops that traverse dynamic data structures. Summarizing, our algorithm to identify if there is any LCD in a loop can be divided into the following steps:

1. The simple pointer statements, S_i, that access the heap inside the loop are annotated with a **Dependence Touch**, `DepTouch`, directive. The Dependence Touch directive comprises three important pieces of information regarding the access to the heap: the **access pointer**, the **access attribute** (ReadS$_i$ or WriteS$_i$, due to the type of access in the S_i statement) and the **access field** (the field of the data structure pointed to by the access pointer which is read or written). For instance, the `S1: aux = p->nxt` statement would have as Dependence Touch directive `DepTouch(p,ReadS1,nxt)`. Note that the Dependence Touch is also identifying the statement in the access attribute.

2. The **Dependence Groups** are created. A Dependence Group, $DepGroup_g$, is a set of access attributes fulfilling two conditions: (a) all the access attributes belong to Dependence Touchs with the same access field, and (b) at least one of these access attributes is a WriteS$_i$.

   In other words, a $DepGroup_g$ states the set of statements in the loop that may potentially lead to a LCD, which happens if they access to the same field of the same memory location in different iterations, and one of the accesses is a write.

   Associated to each $DepGroup_g$, our algorithm initializes a set called $AccessPairsGroup_g$. This set is initially empty but during the analysis process it may be filled with the pairs named **access pairs**. An access pair comprises two ordered access attributes. For instance, a $DepGroup_g$ = {ReadS$_i$,WriteS$_j$,WriteS$_k$} with an $AccessPairsGroup_g$ comprising the pair <ReadS$_i$,WriteS$_j$> means that during the analysis the same field of the same memory location may have been first read by the statement S_i and then written by statement S_j, clearly leading to an anti-dependence. The order inside each access pairs is significant for the sake of discriminating between flow, anti or output dependences.

3. The shape analyzer is fed with the instrumented code. As we have mentioned, the shape analyzer is described in detail in [4] and briefly introduced in Section 2. However, with regard to the LCD test implementation the most important idea to emphasize here is that our analyzer is able to precisely identify at compile time the memory locations that are going to be pointed to by the pointers of the code. With this and with the `DepTouch` directive, the task of the analyzer is to symbolically execute each statement updating the graphs and at the same time, the node pointed to by the access pointer of the statement which is identified in the corresponding `DepTouch`, is "touched". This means, that the memory location is going to be marked with the access attribute of the `DepTouch` directive. In that way, we annotate in the memory location, that a given statement has been read or written in a given field comprised in the location.

   Let's see more precisely how this step works. Each node n of a graph in the RSRSG, has a **Touch Set** associated with it, $TOUCH_n$. When a statement S_j is symbolically executed the access pointer of the statement is going to point to a node, n, that has to represent a single memory location Besides, the associated `DepTouch` directive is also interpreted by the analyzer leading to the updating of the $TOUCH_n$ set. If the Touch set was originally empty we just append the new access attribute AccAttS$_j$ of the `DepTouch` directive. However, if the Touch set does already contains other access attributes, $\{AccAttS_k\}$, two actions take place: first, an updating of the $AccessPairsGroup_g$ associated with the $DepGroup_g$, happens; secondly, the access attribute AccAttS$_j$ is appended to the Touch set of the node, $TOUCH_n = TOUCH_n \cup \{AccAttS_j\}$.

   When updating the $AccessPairsGroup_g$, we check all the access attributes of the statements that have touched previously the node n, i.e., the access attributes that are in the $TOUCH_n$ set. If there is any access at-

tribute, AccAttS$_k$ which belongs to the same $DepGroup_g$ that AccAttS$_j$ (the current statement), then a new access pair is appended to the $AccessPairsGroup_g$. The new pair is an ordered pair <AccAttS$_k$, AccAttS$_j$> which indicates that the memory location represented by node n has been first accessed by statement S_k and later by statement S_j, being S_k and S_j two statements associated to the same dependence group, and therefore a conflict may occur.

4. In the last step, our `LCD_Test` function will check each one of the $AccessPairGroup_g$ updated in step 3. Depending on the pairs comprised by the $AccessPairGroup_g$ we can raise some of the LCDs provide in Fig. 4.

   We note that the `LCD_Test` function must be performed for all the $AccessPairGroup$s created in step 2. When any of the dependence patterns that our test checks, is found on any $AccessPairGroup_g$, then the loop does not contain LCD dependences and can be parallelized.

Our approach improves the scope of applicability of the previously proposed dependence tests to loops that traverse Cycle data structures, even in presence of cyclic traversal patterns. Besides, our method is also able to successfully deal with loops in which the data structures are being modified during the data structure traversal.

4 Locality Analysis

Reference locality is a well-known property that all programs exhibit to some extent. This property states that only a few memory positions are used or reused during a short interval of time, and consequently we could predict which memory locations are going to be accessed in the future. Precisely assuming this property is the basis of common computer hardware design such as cache memory. The goal of such designs is to lessen the growing speed gap existing between the memory system and the processors.

Locality exploitation has become a major issue to deal when obtaining a good efficiency in the execution of a code. In case of regular problems, data is organized as static structures like matrices or arrays, that, in general, generate memory access patterns perfectly known at compile time. Thus the compiler can automatically analyze dependencies and find efficient reordering of data and computations without altering the program semantics.

A more complex case is represented by the named irregular codes. Although these codes work on static data structures like arrays, however content is referenced through indirections. In general, indirect references are unknown at compile time, so they should be fixed at run time. Indirect references may cause loop carried dependences difficult to analyze by the compiler, making locality exploitation a complex task.

Nevertheless an important work has been done with a certain class of irregular patterns commonly found in numerical and scientific codes, as those associated with commutative and associative operators [11, 12], like reductions. Techniques developed in this context prove that it is possible to design effective compiler transformations that exploit locality with no need of solving such indirections at compilation time.

Programs handling pointer-based data structures mean a harder challenge from the point of view of locality exploitation. Usually programmers use pointers when data structures are going to be built and modified during execution. In contrast to regular codes it is not possible to know the memory reference pattern nor the shape of the data structure at compile time, as it is given by the pointer links created at runtime. In addition, data structure components are dynamically allocated as they are needed, and therefore their memory locations may be found scattered over the memory space. The degree of scattering will depend on the allocation policy of new memory locations, which, in general, is not under the programmer's control. Also this fact avoids the possibility of making assumptions about the memory area where data are placed because two different executions of the same program with the same data may generate different memory

```
fun LCD_Test(AccessPairGroup_g)
 if <WriteS_i,ReadS_j> ∈ AccessPairGroup_g
    then return(FlowDep); /* Flow dependence detected between S_i and S_j */
 if <ReadS_i,Write_j> ∈ AccessPairGroup_g
    then return(AntiDep); /* Anti dependence detected between S_i and S_j */
 if <WriteS_i,WriteS_j> ∈ AccessPairGroup_g
    then return(OutputDep); /* Output dependence detected between S_i and S_j */
 if <WriteS_i,WriteS_i> ∈ AccessPairGroup_g
    then return(OutputDep); /* Output dependence detected between S_i and S_i */
 endif
return(NoDep); /* no LCD detected */
```

Figure 4: LCD test

reference patterns.

The effective reordering of data and computations represents also a difficult task in codes handling pointer-based structures. The computation reorganization is limited by the dependence test since if dependences are unknown the order of sentences can not be changed in a safe way. On the other hand, a memory location that is referenced by multiple pointers (aliasing) makes the data reordering process complex because it imply different paths in the data structure that meet in this location.

In this section we analyze the locality exploitation problem in codes using pointer-based data structures. It has been organized as follows. First, we classify this kind of codes regarding several issues to take into account when developing locality exploiting techniques. Next, an overview of different existing approaches exploiting locality in pointer-based programs is presented. Finally, we discuss how shape analysis may be helpful in order to design techniques that are able to improve reference locality in these programs.

4.1 Features of pointer-based codes

Locality exploitation is given to a great extent by the suitable order used to traverse the elements of data structures. Thus, in general, saying that a code exhibit good locality means that the order in which data is placed in memory shows certain affinity with the order data is traversed. In regular codes working on vectors and matrices, this order is determined by the packing used when storing such structures, and the traversing pattern which is generally linear. However, the situation is not so simple in pointer-based codes. In this section we analyze several features related to data ordering and how they influence the strategies to follow.

In a typical code handling pointer-based structures we can distinguish two phases in execution. The first one involves the building of the data structure, and the second one its use. In the process of building the data structure two important components are set, its entry point location in heap memory and the graph that represents the different paths in the structure (given by pointer fields).

One major factor that can determine the strategy to follow is the way that data structure changes dynamically during the program execution. A simple case corresponds to a static structure, that does not change once created. In this situation we can suggest to follow a strategy based on the inspector/executor paradigm. The inspector extract data structure characteristics in runtime, and it carry out transformations oriented to improve locality. However if we are dealing with dynamic structures, that is, structures whose shape is modified, the inspector/executor approach may be not so appropriate. Calling an inspector every time the data structures are modified could cause a high overhead. Nevertheless the inspector approach may be used if inspector is invoked only few times when some modification threshold is surpassed.

The way the data structure is traversed is another

factor to be considered. Initial data locations in memory are determined during the structure creation. We note two facts regarding this initial situation. First, the order in which data is traversed may be different to the order in which it was allocated. Second, the pointer graph may represent various paths to follow but only some of them will be taken when the program runs. A well known example is a tree where we have several typical possibilities to visit the structure elements (preorder, postorder, ...). Traversing information is a major aspect in locality exploitation and hence we need to extract not only information about the structure itself but also additional information about the path followed when the structure is traversed.

In the handling of pointer-based data structures we have, on the one hand, the traversing of the structure until reaching a specific element data and, on the other hand, the computation that is carried out once the above element is located. If most of the time is spent in traversing the data structure, it would be difficult to hide memory latency. The reason is the chain of successive pointers we need to follow to located the target element. If the processing time is small, we can not easily overlap the computation with the prefetching of other elements to be visited, as several indirect references may be involved. Additionally, in problems where computation dominates, the overhead of data structure transformations, like re-layouts, will be less significant, so they can be carried out frequently in the case of dynamic codes.

4.2 Related work

Diverse approaches to improve the locality exploitation and hide memory latency have been explored in the literature in the context of codes handling pointer structures. A complete view of the different locality enhancement issues is discussed in [13]. They propose several locality oriented techniques with the goal of improving the cache memory behavior, based on some reorganization of the dynamic data structures. This can be done at four levels: data structure definition, memory allocation, pointer structure reorganization in runtime and data rearrangement during garbage collection.

Focusing on the three first levels, we find three different strategies: clustering, coloring and compression. *Clustering* means the data structure is packed in such a way that the most likely accessed data elements are placed in the same cache block. For example, in a tree the father and direct descendants can be packed together in the same block. Of course, the final behavior will depend on how the structure is traversed and modified. The aim of *coloring* is avoiding cache conflicts due to the limited associativity of the cache. For this purpose the data elements accessed more often are placed in non-conflicting cache blocks, and those accessed less frequently in the remaining blocks. Finally, *compression* means the data structure is transformed so as several data elements are placed in the same cache block. Now those elements are accessed in a different way, that is, we need to uncompress them to get the original data. For example, a linked list can be compressed in a linear array. In this case, data elements will be referenced by means of an offset instead of following the original chain of pointers.

Placement in memory of the dynamic data structure when created has also an important effect on locality. In [13] a *cache conscious allocator* (`ccmalloc`) is proposed, that allows to allocate memory locations near those that are probably accessed. Since the dynamic data structure may be traversed in different ways, and also it can be modified in runtime, cache conscious allocation techniques can be reinforced by reorganizing the structure layout. A cache conscious layout reorganizer (`ccmorph`) is also described in [13], based on clustering and coloring strategies.

A technique widely used in regular codes (array-based) to tolerate memory latency is data prefetching. Some works analyze this strategy in the context of pointer-based codes [14, 15]. The aim is to solve the known *pointer chasing* problem. In a dynamic data structure elements are accessed after traversing several pointer links. The idea is to determine the location of a data element before needed, trying to hide memory latency. However, this prefetching usually needs several suc-

cessive indirect accesses to memory. To overcome this problem some auxiliary pointers (*jump-pointers*) may be introduced in order to anticipate future elements of the linked structure. In [14] several prefetching jump pointer based approaches are introduced, as the *history prefetching* or *greedy prefetching*. In the first one artificial links are created over the structure by using an historical queue of visited locations. The second one carries out the prefetching of data elements directly linked to an specific element one, not using any artificial jump pointers. In [15] prefetching is done by means of an auxiliary array associated with each data element, containing the location of other data elements that will be visited starting from that particular element.

A limitation of prefetching based methods is the inability of predicting which is the next element of the structure to be visited. In this respect some works [16, 17] try to discover certain regularities in the dynamic reference patterns. Statistical techniques applied to the reference trace can provide these regularity features. After profiling the code and obtaining such features the code can be transformed using this regularity information. Prefetching instructions can be added to the code taking into account which locations have more probability of being referenced in the future. This technique is used in [16] to identify strides in the memory reference pattern, based on the identification of indirect load instructions. In [17] is described an efficient representation of the data reference locality, which is several orders of magnitude smaller than a whole execution trace. It is based on a hierarchical compression algorithm, used to discover regularities in access patterns, called *hot data streams*. After locating these hot data streams, techniques of clustering and prefetching can be applied.

Together with the software techniques to exploit locality, some hardware proposals are found in the literature. In [18, 19] a specific hardware is described that tries to find pairs of load instructions, loads that produce a memory address and loads that consume such an address. These load pairs may correspond with indirect accesses, like those that appear on traversing linked data structures. Load pairs are identified in execution windows and their associated program counter is stored in a table. This correlation table keeps consumer/producer load pairs more recently found. This information is used to prefetch consumer loads.

4.3 Locality exploitation using shape information

In this section we discuss how the shape analysis described in Sec. 2 may become a helpful tool to get useful information for locality exploitation in dynamic data structures. This work is currently in progress and we set out here some key ideas. We focus our attention on data placement in memory, in order to get better use of memory hierarchy. Specifically, two of the previously described approaches are considered: data allocation and data structure rearrangement.

Regarding data allocation we assume that a hierarchy conscious allocator is available. This allocator is able to provide a memory area on demand following certain locality policy, like the proximity to a certain location. The proximity factor depends on the data structure and how elements are inserted in the structure during its creation. It is in this point when the shape analysis may be useful. Let us consider a hierarchy conscious allocator similar to the one described in [13]. This allocator is invoked as `hc-malloc(size, *p)`, indicating the request size and a pointer to define the proximity. We desire that the allocated memory is as near as possible the pointer `p`.

When we convert conventional allocations into hierarchy conscious ones, some knowledge of the structure and code semantics is needed, in order to select which pointer we want to be near to. However this conversion can be easier with shape analysis. Let us consider adding an element at the end of a single linked list as an example. In figure 5 the code for this operation and some illustrative possible shape graphs for different statements are shown. In this figure, RSG1 is the entry RSG to the `addlist()` function. RSG2 is one of the graphs associated with statement S3 in one of the iterations of the loop. Finally, RSG3 is associated with statement S5. Through auxiliary pointers the inser-

tion point is located, adding the new element. In this simple case we desire the new node to be near to the last one. So the hierarchy conscious allocator will need as parameter the position of this last node. We can observe how the shape graph provides this nearness information, and how we can select the pointer to use as argument of the hierarchy conscious allocator.

Regarding the second approach, reorganization of the data structure, the information of shape analysis may also be useful. Let us consider a hierarchy conscious structure reorganizer as it is introduced in [13]. Some of topological parameters required to rearrange the data structure can be obtained from the shape graph. First, we can establish the point of the code where the data structure is completely defined to insert there the call to the reorganizer. Second, it allows us to know if the transformation is applicable, because some restrictions may exist (e.g. valid only for trees). Third, the shape graph can provided some parameters necessary for the reorganizer: entry point to the data structure, maximum number of descendants of a node and the access function to descendants from their father.

An important drawback of data reorganization is due to elements pointed by several other pointers. On transforming the structure if one of these elements changes its original position in memory, all the pointers pointing to must also be changed suitably, in order to hold the structure shape. Precisely the limitations of the reorganizer proposed in [13] are derived from this fact (only one entry point and tree-like restrictions). On analyzing the data structure shape some attributes like *shared* give us just information about nodes potentially referred by several pointers. We can enhance the structure reorganizer to be applicable to a wide range of graphs and not only trees. For this purpose we can associate a pointer list to each element classified as shared. Each pointer in this list act as a *back jump pointer* to the father nodes of this shared element.

In figure. 6 an example is shown. If we take a binary tree its associated shape (a) does not contain any shared node. However if leaf nodes are referenced by several father nodes, then the shape is the same but the attribute shared is asserted for leaves (b). In the case (a) the reorganizer can be easily applied but in (b) several paths can end in a node, making the reorganization difficult or impossible. Shape analysis suggests not only the nodes where back jump pointers can be added but the statements of the code where their values must be determined.

5 Conclusions

The optimization of pointer-based codes is a key issue which has not fully addressed due to the high complexity involved in the process. We believe that shape analysis of dynamic data structures is a powerful tool to help the development of optimizing techniques for this kind of codes. In this paper we have presented how shape analysis may be used to deal with two important topics in high-performance computing, parallelism and locality.

References

[1] R. P. Wilson and M. S. Lam, "Efficient context-sensitive pointer analysis for C programs," in *ACM Conf. on PLDI*, La Jolla, CA, Jun. 1995, pp. 1–12.

[2] M. Shapiro and S. Horwitz, "Fast and accurate flow-insensitive points-to analysis," in *24th Ann. ACM Symp. on PoPL*, Paris, France, Jan. 1997, pp. 1–14.

[3] R. Ghiya and L. J. Hendren, "Putting pointer analysis to work," in *25th Ann. ACM Symp. on PoPL*, San Diego, CA, Jan. 1998, pp. 121–133.

[4] F. Corbera, R. Asenjo and E. Zapata, "A framework to capture dynamic data structures in pointer-based codes," *IEEE Trans. on Parallel and Distributed System*, 15(2), pp. 151–166, Feb. 2004.

[5] L. J. Hendren and A. Nicolau, "Parallelizing programs with recursive data structures," *IEEE Trans. on Parallel and Distributed Systems*, 1, pp. 35–47, Jan. 1990.

[6] S. Hortwitz, P. Pfeiffer and T. Repps, "Dependence analysis for pointer variables," in *ACM Conf. on PLDI*, Portland, OR, July 1989, pp. 28–40.

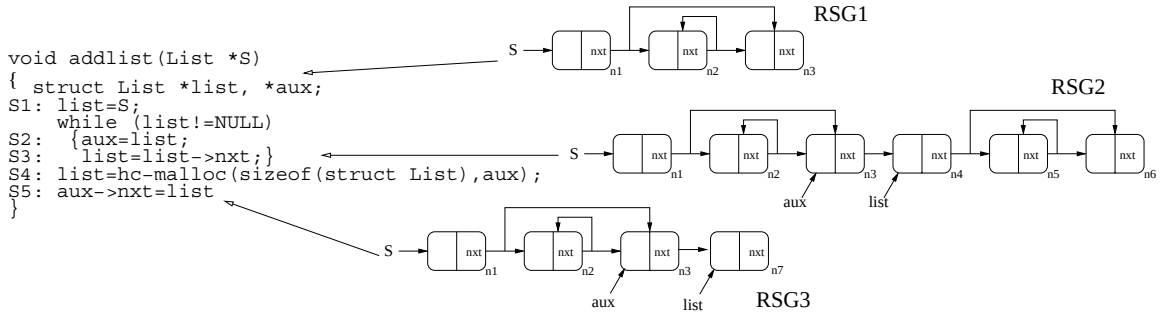

Figure 5: Adding a node at the end of a single linked list

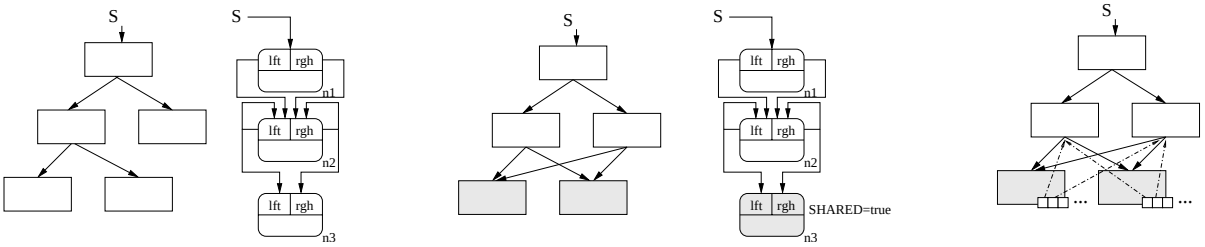

Figure 6: Back jump pointers for shared nodes

[7] J. R. Larus and P. N. Hilfinger, "Detecting conflicts between structure accesses," in *ACM Conf. on PLDI*, Atlanta, GA, Jul. 1988, pp. 21–34.

[8] J. Hummel, L. J. Hendren and A. Nicolau, "A general data dependence test for dynamic, pointer-based data structures," in *ACM Conf. on PLDI*, Orlando, FL, Jun. 1994, pp. 218–229.

[9] R. Ghiya, L. J. Hendren and Y. Zhu, "Detecting parallelism in C programs with recursive data strucutures," in *Int. Conf. on Compiler Construction*, Mar. 1998, pp. 159–173.

[10] Y.S. Hwang and J. Saltz, "Identifying parallelism in programs with cyclic graphs," *J. of Parallel and Distributed Computing*, 63(3), pp. 337–355, 2003.

[11] E. Gutierrez, O. Plata and E. L. Zapata, "A compiler method for the parallelization of irregular reductions on scalable shared memory multiprocessors," in *ACM Int. Conf. on Supercomputing*, Santa Fe, NM, May 2000, pp. 78–87.

[12] H. Han and C.-W. Tseng, "Efficient compiler and run-time support for parallel irregular reductions," *Parallel Computing*, 26(13–14), pp. 1861–1887, 2000.

[13] T. M. Chilimbi, M. D. Hill and J. R. Larus, "Making pointer-based data structures cache conscious," *IEEE Computer*, 33(12), pp. 67–74, Dec. 2000.

[14] C.-K. Luk and T. C. Mowry, "Compiler-based prefetching for recursive data structures," in *ACM Int. Conf. on ASPLOS*, Cambridge, MS, Oct. 1996, 222-233.

[15] M. Karlsson, F. Dahlgren and P. Stenstrom, "A prefetching technique for irregular accesses to linked data structures," in *IEEE Int. Symp. on HPCA*, Toulouse, France, Jan. 2000, pp. 206–217.

[16] Y. Wu, "Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching," *ACM Conf. on PLDI*, Berlin, Germany, Jun. 2002, pp. 210-221.

[17] T. M. Chilimbi, "Efficient representations and abstractions for quantifying and exploiting data reference locality," *ACM Conf. on PLDI*, Snowbird, UH, Jun. 2001, pp. 191–202.

[18] A. Roth, A. Moshovos and G. S. Sohi, "Dependence based prefetching for linked data structures," *ACM Int. Conf. on ASPLOS*, San Jose, CA, Oct. 1998, pp. 115–126.

[19] A. Roth and G. S. Sohi, "Effective jump-pointer prefetching for linked data structures," *26th Ann. ISCA*, Atlanta. GA, May 1999, pp. 111–121.