The handle http://hdl.handle.net/1887/20471 holds various files of this Leiden University dissertation.

**Author:** Holm, Carl Wilhelm Mattias  
**Title:** Optimizing pointer linked data structures  
**Issue Date:** 2013-01-31
Chapter 2

Pointer Structure Restructuring

Predictability in memory reference sequences is a key requirement for obtaining high performance on applications using pointer-linked data structures. This often goes against the dynamic nature of such data structures, as pointer-linked data structures are often used to represent data that dynamically changes over time, which will reduce the predictability, even if the pointer-linked structure was in perfect order initially. Also, different traversal orders of data structures cause radical differences in memory reference behavior when considering the data layout.

Thus, having control on data layout is essential for getting high performance. For example, architectures like the IBM Cell and GPU architectures each have their own characteristics and if algorithms using pointer-structures are to be executed on such architectures, the programmer must mold the data structure in a suitable form. For each new architecture, this means rewriting code over and over again. Another common pattern in code using pointer-linked data structures is the use of custom memory allocators. Drawbacks of this approach are that such allocators must be implemented for various problem domains and that they depend on the knowledge of the programmer, not on the actual behavior of the program. Our restructuring framework is a first step in the direction to liberate the programmer from having to deal with

---

This chapter is based on earlier work done with Harmen van der Spek, so parts of this chapter also appeared in his thesis [50].

COPYRIGHT NOTICE: The original publication is available at www.springerlink.com
domain specific memory allocation and rewriting of data structures.

In this chapter, we present a compiler transformation chain that determines a type-safe subset of the application and enables run-time restructuring of type-safe pointer-linked data structures. This transformation chain consists of type-safety analysis after which disjoint data structures can be allocated from separate memory pools. At run-time, accesses to the memory pools are traced temporarily, in order to gather actual memory access patterns. Next, from these access patterns, a permutation is generated which enables the memory pool to be reordered. Note that these traces are not fed back into a compiler, but are rather used to restructure data layout at run-time without any modification of the original application. Pointers in the heap and on the stack are rewritten if the target they are pointing to has been relocated. After restructuring, the program resumes execution using a new data layout.

Restructuring of linked data structures cannot be performed unless a type-safe subset of an application is determined. This information is provided by Lattner and Adve’s Data Structure Analysis (DSA), a conservative whole-program analysis reporting on the usage of data structures in applications [35, 36]. The analysis results of DSA can be used to segment disjoint data structures into different memory regions, the memory pools. Often, many memory pools turn out to be type-homogeneous, i.e. they store only data of a specific (structured) type. These pools are our starting point.

For type-homogeneous pools, we have implemented structure splitting, similar to MPADS [12], the memory-pooling-assisted data splitting framework by Curial et al. This changes the physical layout of the structures, but logically they are still addressed in the same way (any data access can be characterized by a pool, objectid and field triplet). Structure splitting is not a strict requirement for restructuring, but it simplifies the implementation and results in higher performance after restructuring.

In order to restructure, a permutation vector must be supplied. This permutation vector is obtained by tracing memory pool accesses. Tracing does have a significant impact on performance, so in our framework tracing can be disabled after a memory pool has been restructured. The application itself does not need to be aware of this process at all. It is important to note that tracing and restructuring all happen within a single run of an application.

In order to illustrate the need for restructuring, it is interesting to have a look at what could potentially be achieved by controlling data layout. For this, we used SPARK00 [55, 52], a benchmark set in which the initial data layout can be explicitly controlled. Figure 2.1 shows the potential speedups on an Intel Core 2 system (which is also used in the other experiments, together with its successor, the Core i7) if the data layout is such that the pointer traversals
result in a sequential traversal of the main memory, compared to a layout that results in random memory references. This figure illustrates the potential for performance improvements if data layout could be optimized. Our framework intends to exploit this potential for performance improvements.

Section 2.1 starts with an explanation of work on Data Structure Analysis, that our restructuring framework depends on. Section 2.2 describes the compile-time parts of our framework, while Section 2.3 treats the run-time components. Section 2.4 contains the experimental evaluation of our framework. Restructuring pointer-linked data structures has great potential and in this chapter considerable speedups are shown on the SPARK00 benchmarks. The challenge of SPARK00 lies in closing the performance gap between pointer traversals resulting in random access behavior and traversals resulting in perfectly sequential access behavior. As such, it illustrates the potential, but it does not guarantee that such speedups will be obtained for any application. The overhead of tracing mechanism, which of course does not come for free, is discussed in Section 2.4.2. It is shown that the performance gains do compensate for this overhead within relatively few consecutive uses of the restructured data structure. Restructuring memory pools requires a special stack that can be updated after restructuring. Different mechanisms and their implications are discussed and evaluated in Section 2.4.3. Address calculations need to
be efficient. Therefore, we present improved address calculations, compared to the address calculations in Curial’s work [12], for addressing split memory pools in Section 2.4.4. Related work is discussed in Section 2.5. Future work and conclusions are given in Section 2.6. Part of this chapter has appeared in [54].

2.1 Preliminaries

The restructuring framework presented in this chapter relies on the fact that a type-safe subset of the program has been identified. This is achieved by applying Lattner and Adve’s Data Structure Analysis (DSA) [34, 38, 35, 36]. DSA is an efficient, inter-procedural (whole program), context- and field-sensitive pointer analysis. It is able to identify (conservatively) disjoint instances of data structures even if these data structures show an overlap in the functions that operate on them. Such disjoint data structures can be allocated in their own designated memory area, called a memory pool. We will not describe how DSA works in detail, but we will explain the meaning of the resulting Data Structure Graph (DS Graph) as this forms the basis for our further analyses and transformations.

For implementation and efficiency reasons, data structures are not stored as they have been defined in the original source code. As the DSA provides us with information on type-safety on the whole-program level, it is possible to remap the layout of data structures. This assumes that all uses of such a data structure have been identified and that the data structure cannot escape the program as we know it (otherwise it would not be type-safe).

The pool restructuring framework that we propose in this chapter is based on two techniques: automatic pool allocation and structure splitting. The structure splitting transformations remaps memory pools of records into structured data that is grouped by field instead (essentially, it is mapping from an array of records to a record of arrays). The implementation developed is similar to the MPADS framework of Curial et al. [12], though we optimized the address calculations for commonly occurring structure layouts (Section 2.4.4).

In this section, both DSA and structure splitting, which our analysis passes and transformations depend on, are explained in further detail.

2.1.1 Data Structure Analysis

Data Structure Analysis (DSA) provides information on the way data structures are actually used in a program. First, it is important to understand that
2.1. PRELIMINARIES

```c
int main( int argc, char **argv )
{
    ...
    MatrixPtr tmp = ReadMatrixPtrRow( matrixFile );
    MatrixPtr Matrix = MatrixToFormat( tmp, format );
    ...
    for( i = 0; i < iterations; i++ )
        MatrixMultiplyVec( Matrix, right, result );
    ...
}
```

Figure 2.2: Code excerpt of main function of SPMATVEC.

DSA is *not a shape analysis*. DSA determines which data structures can be proved disjoint in memory. Such a data structure can be a linked list, a tree, a graph or any other pointer-linked data structure.

The result of DSA is the Data Structure Graph (DS Graph). Within this graph, the nodes represent memory objects. A node is described as follows [34]:

Each DS graph node represents a (potentially unbounded) set of dynamic memory objects and distinct nodes represent disjoint sets of objects, i.e., the graph is a finite, static partitioning of the memory objects. Because we use a unification-based approach, all dynamic objects which may be pointed to by a single static pointer variable or field (in some context) are represented as a single node in the graph.

Our primary interest lies in the nodes that are type-homogeneous (all memory objects represented by the node are of the same type and are used in a type-consistent way throughout the entire program.

Construction of the DS graph occurs in three phases. The first is the *Local Analysis Phase* during which the actual program representation is used to construct DS graphs for all functions, taking only local information into account. DS nodes contain flags that indicate whether they contain complete information. The subsequent phase, the Bottom-Up Analysis Phase, combines the information on the local functions with results from their callees, by propagating this information bottom-up. This phase is context-sensitive. The last phase is the Top-Down Analysis phase, which we will not need in our restructuring framework. We use the result from the Bottom-Up Analysis.
Let us illustrate this with an example. Figure 2.2 shows a part of the main function of SPMATVEC, one of the benchmarks used in the evaluation of our method (see Section 2.4, the full sources are available online [51]). Figure 2.3 shows the associated DS Graph. Information about the variables generated by the compilation to the LLVM bit code (which uses an SSA representation) is not shown. The graph shows the two stack variables (specified by the $S$ flag) %tmp and %Matrix. Each of these variables has its own storage space on the stack. Hence the separate nodes. The MatrixFrame structure they are both pointing to is one node, indicating that the analysis cannot prove that they are pointing to disjoint structures. The MatrixFrame structure basically contains three pointers. These are the three arrays of pointers that point to the start of a row, the start of a column and the diagonal elements. The MatrixElement structure is the structure containing the matrix data. It has two self references, that are the two pointers used to traverse the matrix row- and column-wise.

Each function has its own bottom-up DS Graph. Nodes that are related to formal arguments are data structures that are passed in by calling the function. Nodes that do not correspond to a formal argument depict data structures that are instantiated within this function. At this point, such a
node incorporates all information on how this node is used in all callees. The Bottom-Up Analysis ensures that if a node is used in a type-safe fashion this information is propagated to the point where the data structure is instantiated. At that location, a choice can be made about how this data structure instance is treated.

Summarizing, for each data structure, we are interested in the point at which it is actually instantiated and whether it is type-safe in all callees. All such data structures can be stored in a disjoint memory segment, called a memory pool.

### 2.1.2 Automatic Pool Allocation

On top of DSA, Lattner et al. implemented automatic pool allocation [35, 36]. Pool allocation is a transformation that replaces calls to memory allocation functions by custom memory allocators such that disjoint data structures are allocated from disjoint memory regions. This is done by identifying pool-allocatable data structures, as shown in the previous section. After a node in the DS Graph has been determined to be type-safe, all associated memory allocation functions can be identified and be rewritten such that they call a pool allocation library, whose functions take an additional argument, the pool descriptor, that uniquely identifies a data structure instance at run-time. Pool allocated structures allow for precise control on data layout, as it is known that all allocated elements within a particular region have the same type. We use this property to modify the way structure are laid out in main memory.

### 2.1.3 Pool-Assisted Structure Splitting

A useful data layout transformation when a data structure is known to be type-safe is structure splitting. Let us consider a memory pool that only stores elements of a particular structured type. Such a pool is just an array of structures (AOS). If we assume that the size of this array is fixed, the AOS can be easily transformed into a structure of arrays. Figure 2.4 depicts this concept by giving the corresponding structure definitions in C.

Splitting structures has some advantages over normal pool allocation. Firstly, it is possible to do away with all padding which is otherwise needed (except for alignment-imposed restrictions) because primitive data types (i.e. floats, doubles, integers etc..) normally must be aligned to addresses corresponding to the size of the type. In a split structure, however, the elements that follow each other will be of the same type and size. This means that the fields can be packed much more efficiently in the many cases where padding is normally
inserted. Another advantage is that a field in a structure that is not accessed as often as the other elements will not pollute the cache, as unused data will not be taking up cache space.

Structure splitting has its limitations, for example, a split structure will typically be split over multiple memory pages and thus require more active TLB\(^1\) entries. As a consequence of this, a structure that is not used in sequential access (e.g. by following pointer chains), is not likely to yield any performance benefits when split. In addition, when multiple fields of a structure are referenced, the cache efficiency will be worse for split structures than for a non-split structure because in the split version multiple fields will be located in different cache lines, whereas in the original version, those fields are most likely co-located in the same cache line.

The implementation of our structure splitting transformation is similar to the DSA-based implementation of Curial et al. [12], who implemented structure splitting in the IBM XL compiler.

2.2 Compile-time Analysis and Transformation

At compile-time, a whole program transformation is applied in order to rewrite pools to use a split structure layout that supports run-time restructuring. Figure 2.5 shows an overview of the entire compilation chain for our framework. In this section, the analyses and rewriting compiler passes are discussed.

2.2.1 Structure Splitting

Our analysis and transformation chain starts at the point where DSA has been performed on a whole program and pool allocatable data structures have been determined. We then start at the *main* function and traverse all reachable

\(^1\)Translation Look-aside Buffer
2.2. COMPILE-TIME ANALYSIS AND TRANSFORMATION

Figure 2.5: Overview of the pool restructuring compilation chain. GEPI refers to the LLVM GetElementPtrInst instruction.
functions, cloning each function that needs to be rewritten to support the data layout of split structures. Note that cloning is only done along execution paths that are known to have type-safe data structures that can be split safely. Functions are cloned because there might also be calling contexts in which splitting cannot be applied, and these cases must also be dealt with correctly (see Lattner and Adve’s work [35, 36]). From this work, we also use their technique for the identification of the memory pools. It is not possible to split pools that are not type homogeneous because addressing of object fields would become ambiguous and fields of different types and length would introduce aliasing of field values. This information is available from the DSA and pool allocation passes.

During the analysis phase, function clones are generated for split versions of functions and calls are rewritten accordingly. Rewriting of other instructions, such as address calculations are deferred to a later stage, because they are nothing more than a change in the semantics of the address calculation instruction (*GetElementPtrInst*) in LLVM.

Various pieces of information are gathered in the structure splitting analysis pass to be used in subsequent passes. All loads and stores to pool data are identified as well as all loads and stores that store a pointer into a pool. These loads are needed to support the use of *object identifiers* instead of pointers (see Section 2.2.5). The structure splitting pass ensures that all the address calculation expressions (*GetElementPtrInst* in LLVM) whose result points to data in split pools are identified. These expressions must be rewritten before the final code generation at a later stage. The address calculation expressions are not rewritten immediately. Instead, they are rewritten just before code generation because additional passes will need to reason about these expressions.

### 2.2.2 Pool Access Analysis

Pool access analysis is a pass in which all pool accesses (loads and stores) are analyzed. The result of the analysis is that instead of being viewed as an access using a specific pointer, the location read from or written to is represented using a triplet (*pool, object, field*). *Pool* is the pool descriptor used at runtime, *object* the pointer to the object the data belongs to and *field* is the field number that is accessed. Originally, a load or store just used a pointer as its address operand, but now, the more generic notion of pool, object and field can be used. This is analogous to data access in a database (table, row and column).

For each load and store from a split pool, the analysis is performed as
2.2. COMPILATE-TIME ANALYSIS AND TRANSFORMATION

follows:

```c
// Get accessed object
baseObject = get underlying object for accessed object
check that baseObject is also a load

pool = get pool descriptor associated with baseObject

// Get accessed field
gepiInst = get pointer operand of memory instruction
check gepiInst is a GetElementPtr instruction
field = get field index from gepiInst
```

Note that for each access to a pool, it must be possible to determine which field is accessed. This property cannot always be proved if the address of fields is taken, and, therefore, we do not allow that any address of a field is written to any memory location using the LLVM StoreInst. For example, the following C-code snippet will never be restructured:

```c
obj->ptr = &p1->y;
...
*obj->ptr = val;
```

This might be a bit over-conservative, and in a future version, we might define this more precisely. Lattner and Adve’s pointer compression applies the same restriction on field accesses [37].

2.2.3 Stack Management

The primary requirement for structure splitting to work (in terms of code modifications) is the remapping of address calculation expressions so that data is read and written to the relocated location in the split pool. However, if reordering of the pool contents is to be accomplished this is not sufficient. Other pools may for example contain references to the reordered pool (which mean that those references need to be updated). However, these on-heap pointers are not the only references to pool objects that the system needs to deal with. The other type of references that need to be managed are pointers that are stored on the stack and that point into the pool. This problem is
similar to what garbage collectors have to do, and in their terminology, the on-stack pointers are known as roots. Tracking the on-heap pointers can be done by adding additional meta data to the pool descriptor, this meta data is derived from the DSA (that keeps track of connectivity information between pools).

Three different alternatives to accurate stack managing were explored and evaluated. These approaches include explicit pointer tracking, shadow stacks and stack maps. However, only the first method was fully implemented for reasons that will become clear later on. The three different investigated methods for stack management are summarized in Table 2.1.

### Explicit Pointer Tracking

One approach to the stack root issue, is to ensure that all pointers are explicitly tracked at the LLVM level. We call this technique pointer tracking. When a pool descriptor is allocated, a special segment of data is acquired that will be used to track all stack local pointers pointing into the pool, whenever a pointer is allocated on the stack, the location of this pointer is inserted in the per pool stack tracking block. A frame marker is in this case also needed to enable the removal of all the pointer tracking entries associated with a returning function. In LLVM this means that any pointer that is an SSA register must explicitly stored on the stack. The following LLVM function illustrates this a bit further:

---

<table>
<thead>
<tr>
<th>Method</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer Tracking</td>
<td>Simple</td>
<td>Slow</td>
</tr>
<tr>
<td></td>
<td>Portable</td>
<td>Interferes with IR</td>
</tr>
<tr>
<td>Shadow Stack</td>
<td>Fast</td>
<td>Interferes with IR</td>
</tr>
<tr>
<td></td>
<td>Portable</td>
<td></td>
</tr>
<tr>
<td>Stack Map</td>
<td>Fast</td>
<td>Backend Modifications</td>
</tr>
<tr>
<td></td>
<td>No IR Interference</td>
<td>Stack walking not portable</td>
</tr>
</tbody>
</table>

Table 2.1: The three stack management options and their individual advantages and drawbacks
2.2. COMPILATION ANALYSIS AND TRANSFORMATION

The function listed above is transformed into the following:

```c
void @func(pooldesc *pool0) {
  entry:
  bb0:
    %x = load {i32, i32}** %heapObjectAddr
    call void @foo %x
    ret
}
```

In the transformed function the pointer \%x is explicitly backed by a stack variable and this variable is then registered with the run-time function named `split_st_reg_stack_obj`. After the pointer registrations a call to the run-time function `split_st_push_frame` is executed; this function will close the stack frame for the current function in order to speed up the pop operation of the stack. These run-time functions are very short (a few instructions) and will be inlined and thus, do not induce any function calling-overhead. Figure 2.6 shows how the pointer tracking block is constructed during run-time.

In order to reduce this overhead, an approach where stack tracking is disabled in certain functions has been chosen. The pseudo code in the following example illustrates why this is useful:
Pool pool;
Matrix *mtx = readMatrix(pool);

doMatrixOperation(pool, mtx);

PoolRestructure(pool, mtx);

for (int i = 1 ; i < N ; i ++) {
    doMatrixOperation(pool, mtx);
}

Here the critical code is the doMatrixOperation, but if this operation does not call the PoolRestructure function, then this function does not need to track the pointers.

The most important point with the explicit pointer tracking, is that it took a small implementation effort compared to the methods described further on in Sections 2.2.3 and 2.2.3. So, while the method (as shown in experiments later on) is not a good choice for a fielded deployment, it takes very little code to implement both the passes and the run-time support for the explicit pointer tracking.
2.2. COMPILE-TIME ANALYSIS AND TRANSFORMATION

Shadow Stacks

The second approach that we investigated for tracking pointers on the stack, was the utilization of a shadow stack. This technique is based on the garbage-collection method described by Henderson [23]. To implement shadow stacks the compiler creates a per function data structure where pointers that are stored on the stack will be stored as a group, such that each pointer can be addressed relative to the base of this data structure. When a function is called, such a structure is allocated on the stack and this structure is then registered with the runtime. This pre-registration cuts down on the additional registration overhead compared to the pointer tracking, by only inferring one registered pointer per function call.

Stack Maps

The third alternative is the construction of stack maps (for example described by Agesen in [1]). Stack maps are structures that are generated statically for each function. These structures describe the stack frames of the corresponding functions. The maps are computed during the code generation phase and contain information about, for example, frame-pointer offsets of the pointers allocated by the function. The main advantage of delaying this to the code generation phase is that the transformation will not interact in any way with earlier optimizations. The main drawback is that the stack walking will become platform dependent and this may not necessarily suit every compiler.

For non-split structures, the derived pointers to fields in the structures can easily be computed by adding a constant offset to the base pointer of the structure. For split structures, however, this is not possible anymore. In a split structure the field addresses no longer have constant offsets from the base pointer of the structure (see Figure 2.7 for a graphical explanation of why this happens).

2.2.4 Address Calculations

It is obvious that calculating addresses for the fields in the structures must be very efficient. This fact was already stressed by Curial [12], but he did not optimize the address calculation expressions and their selection rules to the same extent as we did. If this calculation is inefficient, it potentially nullifies much of the performance improvement gained from the more cache-efficient split structure representation. In general, the offset for field \( n \) can be represented by the following equation:
\[ \text{offset}_n = k_n + \text{sizeof}_n \frac{p \& (\text{sizeof}_\text{pool}_n - 1)}{\text{sizeof}_0} - p \& (\text{sizeof}_\text{pool}_n - 1) \] (2.1)

where \( k_n \) is the constant offset to field array \( n \) from the pool base. The sub-expression \( p \& (\text{sizeof}_\text{pool}_n - 1) \) calculates the object pointer \( p \)'s offset from the pool base and the expression \( \frac{p \& (\text{sizeof}_\text{pool}_n - 1)}{\text{sizeof}_0} \) calculates the object index in the pool.\(^2\)

When the accessed field is the first field of the structure then \( \text{offset}_0 = 0 \) and if the size of the accessed field is the same as the first field of the structure then \( \text{offset}_n = k_n \).

We have observed that in many common cases the size difference between the accessed field and the first field is a power of two. Taking this observation into account, we introduce two additional expressions. When the size of the accessed field is greater than the first field of the structure we have that:

\[ \text{offset}_n = k_n + (\text{sizeof}_n - \text{sizeof}_0) \frac{p \& (\text{sizeof}_\text{pool}_n - 1)}{\text{sizeof}_0} \] (2.2)

and when the size of the first field is greater than the accessed field use the following expression:

\[ \text{offset}_n = k_n - (\text{sizeof}_0 - \text{sizeof}_n) \frac{p \& (\text{sizeof}_\text{pool}_n - 1)}{\text{sizeof}_0} \] (2.3)

Equation 2.1 can be viewed as adding the pool base to the offset from the address of the \( n^{th} \) field of the first object, see Figure 2.7. This figure also demonstrates that Equation 2.2 and 2.3 take into account the linear drift of field \( n \) due to the size differences between fields 0 and \( n \), with respect to the object’s pool index and the constant offset \( k_n \).

It is assumed that further passes of the compiler will apply strength reduction on all multiply and divides involving a power of two constant. Fog [15] gives the cost for various instructions for a 45nm Intel Core 2 CPU. These numbers have been used to estimate the cost in cycles for the various equations calculating the offsets. Assuming that the expressions have been simplified as much as possible through, for example, constant folding and evaluation, we get that when neither \( \text{sizeof}_0 \) nor \( \text{sizeof}_n \) are powers of two, Equation 2.1 will take 26 cycles. If \( \text{sizeof}_0 \) is a power of two the same equation will take 6 cycles (as

\(^2\)Note that in this context, & is the C-operator for a bitwise AND.)
the very costly divide will be reduced to a shift) and if both sizes are powers of
two it will take 4 cycles. Equation 2.3 will take 3 cycles, and Equation 2.2 will
take 3 cycles in the normal case (or 2 cycles if $\text{sizeof}_n - \text{sizeof}_0 = \text{sizeof}_0$).

The address calculations as defined by Equations 2.1 and the elimination
of calculations if accessing the first field are already used in MPADS [12], but
our additional Equations 2.2 and 2.3, have some important properties. They
allow the calculation of the field offsets to be reduced to 2 or 3 instructions
instead of 4, as the code generator will merge the divide and the multiplication
operation into a single shift operation and that the third term in Equation 2.1
has been eliminated. Note that for Equation 2.2 when $\text{size}_n - \text{size}_0 = \text{size}_0$,  

Figure 2.7: Graphical representation of split pools and the field offset expres-
sions detailed in Section 2.2.4 for a split pool consisting of structures with
elements of sizes 2, 4 and 1 bytes. Each shade of gray represents an individual
object. The object pointer $p$ is in this case is pointing at the third object and
the derived pointer $p_n$ is pointing to the second field of the third object.
LLVM will automatically eliminate the multiply and the divide instruction, giving even more savings.

The most notable equation cost (26 cycles) come from the existence of a divide instruction in the expression. This will, for example, happen when the first field of a structure is an array of three 32-bit values (arrays are not split since they are already sequential) and the next element is a 32 or 64 bit value. In those cases up to 23 cycles may be saved on the address calculation because the divide instruction has been eliminated through strength reduction introduced by Equation 2.3.

Overall it can be said that a compiler that splits structures should also reorder the fields in a structure so that address calculations are made as simple as possible. For example, if a structure contains three fields of lengths 1, 2 and 4 bytes, then the field ordering should place the 2-byte element first under the condition that the access frequency of the fields is the same. Though, at this moment our implementation does not do this and this field reordering remains on the future work list.

2.2.5 Converting Between Pointers and Object Identifiers

Instead of storing pointers in split memory pools, object identifiers are used. Object identifiers can be used in type-homogeneous pools to uniquely identify an object within a pool. As shown in Section 2.2.2, together with a field number, each data element can be addressed. Object identifiers are a more compact representation than pointers and also more compact than byte offsets from a pool base pointer, as used in Lattner and Adve’s static pointer compression [37]. Their dynamic pointer compression transformation also uses object identifiers. In that case, it provides a representation independent of the size of fields, whereas byte offsets would need to be rewritten if field sizes change.

Our motivation to use object identifiers is different. While our framework would also benefit from pointer compression (currently object identifiers are stored as 64-bit unsigned integers), we use object identifiers because they can be used as indices in permutation vectors and because they provide position independence for data structures. For future developments, the object indexing will aid in using data structures in hybrid architectures and environments because the representation is position-independent. Whenever a pointer is loaded from memory, the conversion is done in an architecture and context-dependent way.
2.2. **COMPILE-TIME ANALYSIS AND TRANSFORMATION**

```c
uintptr_t
ptr_to_objid(split_pooldesc_t *pool, void *obj)
{
    uintptr_t objIdx;
    if (obj == 0) {
        return 0; // Special case: NULL pointer
    } else {
        uintptr_t poolBase = (uintptr_t)pool->data;
        uintptr_t objOffset = poolBase - (uintptr_t)obj;
        objIdx = objOffset / sizeof_field(0);
    }
    return objIdx;
}
```

Figure 2.8: Store Value Rewrites

Section 2.2.2 described how all loads and stores to memory pools can be represented as a *(pool, object, field)* triplet. In the case that *field* is a field that is pointing to pool-allocated data (whether this defines a recursive data structure or a link to another data structure does not matter), the pointer value that will be stored into the memory pool needs to be converted to an object identifier before it is stored. When such a pointer value is loaded from a memory pool, it must be converted from an object identifier to a pointer. Loads and stores to the stack are unaffected and thus will contain real pointers. As no pointers to fields, but only pointers to objects will be stored to the memory pools, we only need conversion functions for object pointers. For store instructions, the value to store is rewritten as illustrated in Figure 2.8 and for load instructions, the loaded value is rewritten as illustrated in Figure 2.9.

Note that the actual implementation uses LLVM bit code and uses a bitmask instead of an if-statement to handle the NULL pointer.

Compared to the description of object indexing used in the pointer compression transformation by Lattner and Adve [37], our implementation differs in some ways. In their work, object indices are not only present in the heap, but are also used on the stack and in LLVM's virtual registers. Pointer comparisons and assignments do not need the object identifier to be expanded to a full pointer in their framework. In our framework, only loads and stores of pointers (only to pool objects) to split pools need rewriting, and the rest of the code will run unchanged. It also simplifies the restructuring step: on the heap, we only need to handle object identifiers, on the stack we only have to
void* objid_to_ptr(split_pooldesc_t *pool, uint64_t *objIdx)
{
    if (objIdx == 0) {
        return 0; // Special case: NULL pointer
    } else {
        uintptr_t poolBase = (uintptr_t)pool->data;
        uintptr_t objOffset = objIdx * sizeof_field(0);
        uintptr_t obj = poolBase + objOffset;
        return (void *)obj;
    }
    } Figure 2.9: Load Value Rewrites
deal with full pointers.

2.2.6 Restructuring Instrumentation

Pool tracing and restructuring of data structures requires instrumentation of the code with calls to the tracing run-time. During pool access analysis, all loads and stores to pools have been identified and are represented using the triplet (pool, object and field). All these instructions can be instrumented such that a per pool, per field trace of object identifiers is recorded. Currently, we only trace load instructions.

We only enable tracing for one execution of a function and its callees because tracing is a method that does not come for free. After this first tracing, the data is restructured and tracing is disabled. This is accomplished by generating two versions of the function, one with and one without tracing. Selecting the proper function is done through a global function pointer that is set to the non-traced version after a trace has been obtained.

2.3 Run-time Support

Extracting a type-safe subset of the program and replacing its memory allocation by a split-pool-based implementation requires run-time support, similar to the run-time provided for regular pool allocation. The split-pool runtime provides create and destroy functions to split pools and for memory allocation
and deallocation functionality. In addition, some common operations implemented in the standard C library are also provided, such as `memcpy` (which needs to copy data from multiple regions due to the split layout), thereby widening the applicability of the framework.

In this section, the run-time system for splitting and restructuring is described. While this run-time system has been implemented specifically to support our pool restructuring framework, it can also be used as a standalone library, giving the user the ability to explicitly use split and restructurable data structures. Note that pool connectivity must be explicitly specified if the library is used separately from the compiler in order to keep data structures consistent after restructuring.

### 2.3.1 Application Programming Interface

The split-pool run-time offers implementations for initializing pools and for memory allocation and deallocation. Tables 2.2 and 2.3 describes the run-time functions needed to support restructuring of split pools.

### 2.3.2 Tracing and Permutation Vector Generation

In order to restructure a memory pool a permutation must be supplied to the restructuring run-time. The pool access analysis pass (Section 2.2.2) provides the compile-time information (pool, object and field) about all memory references and these memory references can all be traced. Traces are generated per pool, and per field. For each pool/field combination, this results in a trace of object identifiers. From any of these traces, a permutation vector can be derived which can be used to permute a pool. The permutation vector is currently computed by scanning the trace sequentially and appending the object identifiers encountered to the vector, avoiding duplicates:

```cpp
perm[0] = 0;
permLen = 1;
for (i = 0; i < maxTraceEntry; i++) {
    if (!perm[trace[i]]) {
        perm[trace[i]] = permLen;
        permLen++;
    }
}
```
Chapter 2. Pointer Structure Restructuring

### Table 2.2: Functions of the split pool run-time with restructuring support.

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Allocate Memory from a Split Pool</strong></td>
<td>split pool desc, t *pool, uintptr t obj, cnt uint32 t n s p l i t obj len, tf i e l d cnt</td>
<td>Allocates memory from a split pool creation and initialization. This is a replacement for malloc. Non-split objects length and split object length are both specified. The number of objects allocated is NumBytes, which is followed by a list of integers specifying the size of each field in bytes.</td>
</tr>
<tr>
<td><strong>Destroy</strong></td>
<td>split pool desc, t *pool</td>
<td>Destroys a pool and frees up all memory.</td>
</tr>
<tr>
<td><strong>Allocate Memory</strong></td>
<td>split pool desc, t *pool, NumBytes unsigned</td>
<td>Allocates an integer number of objects from a pool. This is a replacement for malloc.</td>
</tr>
</tbody>
</table>

The number of objects allocated is NumBytes. Non-split objects length and split object length are both specified. The number of objects allocated is NumBytes, which is followed by a list of integers specifying the size of each field in bytes.
<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments</th>
<th>Return value</th>
<th>Description</th>
</tr>
</thead>
</table>
| split_poolrealloc     | • split_pooldesc_t *pool  
• void *obj  
• unsigned NumBytes                                                | void *       | **Reallocate Memory from a Split Pool.**  
Replacement for `realloc` in the standard C library.                                                |
| split_poolfree        | • split_pooldesc_t *pool  
• void *obj                                                               | void         | **Free Pool Allocated Objects.**  
Replacement for `free` in the standard C library.                                                   |
| split_pooltrace_init  | • split_pooldesc_t *pool                                                | split_pooltrace_info * | **Initialize tracing for a pool.**  
Initializes tracing data structures for a pool.                                                      |
| split_pooltrace_trace_base_stack | • void *pool  
• uint32_t field  
• void *ptr                                                      | void         | **Adds an entry to the trace for a specific field of a pool.**  
Two versions exist, one for pointers on (not to!) the stack that are dereferenced, one for pointers on the heap.  
This is done because pointers on the heap are stored as object identifiers and pointers on the stack are stored as full pointers and thus need to be converted to an object id first when adding an entry to a trace. |
| split_pooltrace_trace_base_heap | • void *pool                                                    |              |                                                                                                  |

Table 2.3: Functions of the split pool run-time with restructuring support.
Element 0 is reserved to represent the NULL pointer and is thus never permuted.

Tracing does not come for free and, therefore, tracing should be avoided if it is not necessary. For the evaluation of our restructuring method we choose to trace the first execution of a specified function (compiler option specifies which function), restructure using this trace and then disable tracing. In a future implementation, this will be dynamic and tracing could be triggered if a decrease in performance is detected (for example by using hardware counters).

### 2.3.3 Pool Reordering

One of the more important parts of our system is the pool-rewriting support. Rewriting in this context means that a pool is reordered in memory, so that it is placed in a more optimal way with respect to memory access sequences. This is done during run-time, and the re-writing is based on passing in a permutation vector generated during run-time as described in Section 2.3.2. We have implemented a copying rewriting-system that uses permutation vectors that specify the new memory order of the pool. Although permutation vectors could technically be generated during compile time in some cases where data is not input dependent.

When a permutation vector is available, a pool can be rewritten in order to optimize the memory layout. The pool rewriting algorithm that we have devised has three distinct phases:

1. *Pool rewrite*, where the actual pool-objects are being reordered

2. *Referring pool rewrite*, where pointers in other pools that refer to the rewritten pool are updated to the new locations

3. *Stack update*, where the on-stack references to objects in the rewritten pool are updated

The basic algorithm for the interior pool update is as follows:
newData = mmap(pool.size);
foreach field in pool {
    foreach element in field {
        if field contains recursive pointers {
            newData[field][permVec[element]] = permVec[pool.data[field][element]];
        } else {
            newData[field][permVec[element]] = pool.data[field][element];
        }
    }
}
munmap(pool.data);
pool.data = newData;

In this case each field in the split pool is copied into the new address space, and relocated according to the permutation specified in the permutation vector. If the value in the field is itself a pointer to another object in the pool, that pointer is remapped to its new value.

For the second phase where all the referring pools are updated, the rewrite is even simpler:

for (referrer in pool.referrers) {
    for (ent in referrer.field) {
        referrer.field[ent] = permVec[referrer.field[ent]];
    }
}

Here, each pool that refers to the rewritten pool will have the field containing those pointers updated with the new locations.

The algorithm detailed here assumes that each pool descriptor has information available regarding the pool connectivity (i.e. which fields in other pools that points out objects in the rewritten pool). This information can be derived from the DSA discussed earlier. This connectivity information is therefore registered as soon as the pool is created.
2.3.4 Stack Rewriting

As already discussed in Section 2.2.3, the program stack is managed through explicit pointer tracking. When a pool descriptor is allocated, a special segment of data is acquired that is used to track all pointers on the stack pointing into the pool. Whenever a pointer is allocated on the stack, the location of this pointer is inserted in the per-pool stack-tracking block.

When a pool is rewritten, the current stack will be traversed and all base and derived pointers to locations within the pool are rewritten to reflect the new location of the object. This block makes a distinction between base pointers and derived pointers, and each derived pointer is also tagged (in the stack tracking block) with the field to which it refers.

2.4 Experiments

The challenge of a restructuring compiler is to generate code that will automatically restructure data, either at compile-time or run-time, in order to achieve performance that matches the performance when an optimal layout would be used. In the introduction the potential of restructuring was shown by comparing execution of the benchmarks using explicitly defined data layouts. In the experiments here, we ideally want to obtain similar performance gains, but by automatic restructuring of data layout of the used pointer-linked data structures.

We use the benchmark set SPARK00 which contains pointer benchmarks whose layout can be controlled precisely [55, 52]. The pointer-based benchmarks used are: SPMATVEC (sparse matrix times vector), SPMATMAT (sparse matrix times matrix), DSOLVE (direct solver using forward and backward substitution), PCG (preconditioned conjugate gradient) and JACIT (Jacobi iteration).

These benchmarks store their matrix using orthogonal linked lists (elements are linked row-wise and column-wise). All of them traverse the matrix row-wise, except DSOLVE, which traverses the lower triangle row-wise and the upper triangle column-wise.

For all benchmarks, one iteration of the kernel is traced, after which the data layout is restructured. After this, tracing is disabled. This all happens at run-time, without any hand-modification the application itself.

The experiments have been run on two platforms. The first is the Intel Core 2 platform, an Intel Xeon E5420 2.5 GHz processor with 32 GiB of main memory, running Debian 4.0. The other system is an Intel Core i7 920 2.67GHz
2.4. EXPERIMENTS

based system with 6 GiB of main memory, running Ubuntu 9.04.

2.4.1 Pool Reordering

As shown in the introduction, being able to switch to an alternative data layout can be very beneficial. We applied our restructuring transformations to the SPARK00 benchmarks and show that in ideal cases, speedups exceeding 20 are possible by regularizing memory reference streams in combination with structure splitting. Of course, the run-time introduces a considerable amount of overhead and is a constant component in our benchmarks. We will consider this overhead separately in Section 2.4.2 to allow a better comparison between the different data sets.

As a first step in our experimentation, we first determine the maximal improvement possible, by using an initial layout that causes random memory access. Figure 2.10(a) and 2.10(b) show the results of restructuring on the pointer-based SPARK00 benchmarks (except DSOLVE, which is treated separately), if the initial data layout causes random memory access, on the Intel Core 2 and Core i7, respectively. The data set size increases from left to right. As shown in previous work [55, 52], optimizing data layout of smaller data sets is not expected to improve performance that much and this fact is reflected in the results. On both architectures, restructuring had no significant effect for data sets fitting into L1 cache. These sets have not been included in the figures. For sets fitting in the L2 and L3 cache levels, speedups of 1 – 6× are observed. The Core i7 has a 8 MiB L3 cache, whereas the Core 2 only has two cache levels. This explains the difference in behavior for the matrix Sandia/ASSIC_100ks, which shows higher speedups for the Core 2 for most benchmarks. However, it turns out that the Core i7 runs almost 3× faster when no optimizations are applied on SPMATVEC for this data set. Therefore, restructuring is certainly effective on this data set, but the greatest benefit is obtained when using data sets that do not fit in the caches.

An interesting case is DSOLVE, in which the lower triangle of the matrix is traversed row-wise, but the upper triangle is traversed column-wise. As the available data layouts of the matrices are row-wise sequential (CSR), column-wise sequential (CSC) or random (RND), none of these orders matches the traversal order used by DSOLVE. Figure 2.11(a) and 2.11(b) shows the results for DSOLVE using the different memory layouts on the Core 2 and Core i7, respectively. The matrices are ordered differently than in the other figures, as DSOLVE uses LU-factorized matrices as its input, which have different sizes depending on the number of fill-ins generated during factorization. The matrices have been ordered from small to large (in the case of DSOLVE, this
Figure 2.10: Speedups obtained using restructuring on the SPARK00 benchmarks. The initial data layout is random.
2.4. EXPERIMENTS

Figure 2.11: Speedups obtained using restructuring on DSOLVE for all different initial layouts. Input data sets are ordered by size (after LU-factorization).
is the size after LU-factorization).

For the lung1 data set, a decrease in performance is observed, but for the larger data sets, restructuring becomes beneficial again. Speedups of over $6 \times$ are observed for the Core i7, using CSC (column-wise traversal would yield a sequential memory access pattern) as initial data layout. In principle, the RND (initial traversal yields a random memory reference sequence) data set could achieve much higher speedups if after restructuring the best layout has been chosen. Currently, this is not the case for DSOLVE and we attribute this to the very simple permutation vector generation algorithm that we use (see Section 2.3.2). Generation of permutation vectors from traces will be improved in future versions of the framework.

### 2.4.2 Tracing- and Restructuring Overhead

Our framework uses tracing to generate a permutation vector that is used to rewrite the memory pool. Traces are kept for each field of a pool and one of these traces is used for restructuring. Currently, the trace to be used is specified as a compiler option, but this could potentially be extended to a system that autonomously selects an appropriate trace. This will be addressed in a forthcoming paper.

Tracing and the subsequent restructuring step have an impact on the performance. One cannot simply trace everything all the time as the system will run out of memory very quickly. In the benchmarks, we choose to only trace the first iteration of the execution of the kernel. In order to minimize the overhead of the tracing, the trace will only contain object identifiers, as described in Section 2.3.2. So for instance, if a linked list contains a floating point field and this list is summed using a list traversal, then if both the pointer field and the floating-point field are traced there is an overhead of 2 trace entries per node visited. In our experiments, the structure operated on is 32 bytes and tracing the above-mentioned traversal would add 16 bytes per node extra storage requirements when using 64-bit object identifiers. Using 32-bit objects identifiers, this would be reduced to 8 bytes. Subsequently, the memory pool is restructured using the information of the trace which relates to the field that contains the floating point values of the linked list nodes.

The overhead of the tracing and restructuring has been estimated by running a single iteration of each kernel with and without tracing and restructuring enabled, using a data layout causing random memory access. Figure 2.12 shows the interpolated execution times of the benchmark PCG, both with and without restructuring for the Core 2 and Core i7 architectures. The initial data layout produces random memory access behavior of the application, which is
2.4. EXPERIMENTS

Figure 2.12: Execution times with and without restructuring. The break-even points are marked with a dot.
Table 2.4: Number of iterations for the break-even points when tracing and restructuring is enabled, and when using an initial random data layout. The matrices are ordered by increasing size. The lower part of the table contains the larger data sets, which do not fit in the caches. DSOLVE performs worse when hung, therefore a break-even point is not applicable. The missing entries for JACIT are due to zero elements on the diagonal. DSOLVE performs worse when hung, therefore a break-even point is not applicable. The missing entries for JACIT are due to zero elements on the diagonal.

<table>
<thead>
<tr>
<th></th>
<th>lung1</th>
<th>42.1</th>
<th>51.8</th>
<th>113.9</th>
<th>58.3</th>
<th>388.5</th>
<th>31.5</th>
<th>98.8</th>
<th>55.4</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>bcsstm34</td>
<td>24.3</td>
<td>6.2</td>
<td>53.6</td>
<td>29.5</td>
<td>22.7</td>
<td>5.6</td>
<td>27.2</td>
<td>6.7</td>
<td>19.8</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>cage9</td>
<td>21.0</td>
<td>8.1</td>
<td>44.3</td>
<td>26.1</td>
<td>22.1</td>
<td>8.1</td>
<td>28.6</td>
<td>10.3</td>
<td>3.1</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>rdist3a</td>
<td>17.9</td>
<td>5.6</td>
<td>39.5</td>
<td>21.1</td>
<td>17.7</td>
<td>5.2</td>
<td>2.4</td>
<td>5.6</td>
<td>2.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jan99jac040</td>
<td>16.0</td>
<td>8.0</td>
<td>16.3</td>
<td>15.3</td>
<td>17.8</td>
<td>8.2</td>
<td>2.4</td>
<td>5.6</td>
<td>2.4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>crystm01</td>
<td>8.3</td>
<td>4.9</td>
<td>17.1</td>
<td>17.0</td>
<td>9.1</td>
<td>4.9</td>
<td>2.6</td>
<td>4.7</td>
<td>2.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASIC100ks</td>
<td>8.3</td>
<td>4.9</td>
<td>17.1</td>
<td>17.0</td>
<td>9.1</td>
<td>4.9</td>
<td>2.6</td>
<td>4.7</td>
<td>2.6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>matrix</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dsolve</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jacit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>poa</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>spmdmat</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>spmdvec</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>g2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N/A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>matrix</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For DSOLVE, the missing entries are due to matrices that take too long to factorize.
2.4. EXPERIMENTS

eliminated after the first iteration when tracing and restructuring is used. After the first iteration, the application switches automatically to the non-traced version, which uses the restructured data. Four different matrices have been used which are representative in terms of performance characteristics (see Figure 2.10(a) and 2.10(b)). The break-even points for all matrices are included in Table 2.4.

The figures show that tracing does come with an additional cost, but for most (larger) data sets the break-even point is reached within only a few iterations. For instance, for all data sets shown in Figure 2.12, the break-even point is reached within 4 iterations, except for cage9, which is the smallest data set depicted. Interestingly, on the Core i7, the break-even point is reached even quicker, making restructuring more attractive on this architecture.

Although we have shown in this section that the additional costs of tracing are manageable, it should be noted that we only showed this on computational kernels. In general, it is not recommended to trace a full application code. Therefore, as we have noted earlier in this chapter, tracing should be turned on explicitly by a compiler option and coupled with a specification of the functions that should be traced.

2.4.3 Run-time Stack Overhead

In order to quantify the overhead from the stack management that is needed if pool restructuring is desired, a few custom programs have been written. The interesting overhead in this case will be a measurement of per-function and per-pointer overhead.

In order to measure this overhead an experiment was carried out where a function is called that declares (and links together) a certain number of pointers that point into a pool. This was repeated for a multiple number of pointers and for both a version of the program built without the semi-managed stack and one version that was built with the semi-managed stack enabled. The function was in term executed a certain number (over a million) of times.

The following code demonstrates how this experiment was conducted:
CHAPTER 2. POINTER STRUCTURE RESTRUCTURING

```c
listelem_t*
nextElem(list_t *list)
{
    if (list->current)
        list->current = list->current->next;

#pragma MAKE_POINTERS
    return list->current;
}
```

where the MAKE_POINTERS pragma was replaced by:

```c
listelem_t *a0 = list->current;
listelem_t *a1 = a0;
listelem_t *a2 = a1;
listelem_t *a3 = a2;
...n
listelem_t *aN-1 = aN;
```

The execution time for the loop calling the `nextElem` function was measured and the difference between the managed version and unmanaged version should thus represent the overhead introduced for that number of pointers in the given number of calls to the function.

Figure 2.13 shows the execution time on a 2.5GHz Intel Core 2 Duo, of 4 million calls to the function above in several runs with different numbers of pointers declared and used in one function. The data evaluates to a base cost of 5 cycles per pointer being linked, for the pointer tracking alternative the cost is around 27 cycles per pointer being registered and linked. This gives the penalty of explicit pointer-tracking to 22 cycles per pointer being tracked. This overhead is obviously quite substantial, but the compilation chain described in this chapter employed a simple optimization in order to minimize the overhead.

The optimization used was based on disabling the pointer tracking when not needed, for example in descendant functions from the one that calls the restructuring run-time (since the stack on the descendants will be dead anyhow, when the restructuring function is invoked).
Figure 2.13: Execution time of a function with different stack management approaches

Since the shadow stack and stack map strategies have not been implemented and thus these strategies have not been evaluated using compiler generated code, a hand-written implementation of these strategies has been used to estimate the overhead of these techniques.

By pooling all the pointers associated with a pool in a function into a single per-function data structure, it is possible to eliminate all per-pointer overhead
associated with registering each pointer. In this case, only the address of the record containing all the pointers needs to be registered. This has its own problems, as it prevents certain optimizations such as the elimination of unused pointers (though the pointer tracking suffers from the same issue).

The stack map approach offers none of the run-time overhead (except during the stack walks when program counter entries on the stack are translated into function ids), but does on the other hand require modifications in the compiler’s backend.

2.4.4 Address Calculations

The address calculation expressions used are an improved variant of those introduced by Curial et.al. [12]. These improvements have been verified experimentally by running two versions of the pointer-based applications from the SPARK00 benchmark suite [55, 52], one with the new optimized address calculation expressions enabled, and one version with only the general addressing equations used by MPADS enabled. It should be noted that the implementation described in this chapter is not using the same compiler framework as MPADS which is based on XLC. Thus a direct comparison between Curial’s work and the compiler chain introduced in this chapter has not been carried out.

The matrix input files are sparse and inserted in row-wise order, leading to a regular access pattern upon traversal. In Figure 2.14, the matrices are ordered by size. For the SPMATMAT benchmark the same matrices are used three times each: one pass using one column of the right-hand side matrix, the second pass using seven columns and the third pass using 30 columns. Note that the matrix multiplication in SPMATMAT is multiplying a sparse matrix with a dense matrix. The result of this multiplication is a dense matrix.

SPARK00 was compiled with LLVM GCC in order to generate LLVM bit code. The bit code was then passed through the LLVM-linker and the pool allocation and structure splitting optimization passes.

When running the experiments, it was expected that the new field offset equations will in principle never be less efficient than the generic ones, excluding effects on instruction caches and any reordering that the compiler may or may not do due to the changed instruction stream.

Table 2.5 gives the average improvements of the addressing optimizations. In Table 2.5, the SPMATVEC benchmark actually lost in performance, this was due to instruction cache conflicts in the new code. Figure 2.14 shows the general behavior of the benchmarks where the relative performance improvements is greater for smaller data set sizes. This is because the new instruction
2.5. RELATED WORK

Optimization of data access in order to improve performance of data-intensive applications has been applied extensively, either by automatic transformations or by hand-tuning applications for efficient memory access. In some cases, memory access patterns can be determined symbolically at compile-time. In such cases, the traditional transformations such as loop unrolling, loop fusion or fission and loop tiling can be applied. For applications using pointer-linked data structures, such techniques can in general not be applied.

The traditional methods mentioned above change the order of instruction execution such that data is accessed in a different way, without affecting the result. One might as well change the underlying data layout, without affecting the computations. This is exactly what has been done on pointer-linked data structures in this chapter.

In order to be able to automatically control the layout within type-unsafe languages such as C, a type-safe subset must be determined. The Data Structure Analysis (DSA) developed by Lattner and Adve does exactly that [35, 36]. It determines how data structures are used within an application. This has been discussed in Section 2.1.1.

DSA should not be confused with shape analysis. Shape analysis concerns the shape (e.g. tree, DAG or cyclic graph) of pointer-linked data structures. Ghiya and Hendren proposed a pointer analysis that classifies heap directed

<table>
<thead>
<tr>
<th>Bench Name</th>
<th>Address Calc Improvements</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSOLVE</td>
<td>4.87 %</td>
</tr>
<tr>
<td>JACIT</td>
<td>4.59 %</td>
</tr>
<tr>
<td>PCG</td>
<td>1.99 %</td>
</tr>
<tr>
<td>SPMATMAT</td>
<td>3.81 % (6.22%/4.16%/1.05%)</td>
</tr>
<tr>
<td>SPMATVEC</td>
<td>-6.11 %</td>
</tr>
</tbody>
</table>

Table 2.5: Performance gain averages in percent for pool allocation and the improved field offset equations. Note that SPMATVEC has a negative improvement due to instruction cache conflicts. For SPMATMAT, different figures are given in parentheses for 1, 7 and 30 columns in the right hand matrix mixture actually plays a greater part in those cases. For the larger data sets, the performance is more bounded by the memory latency and thus the instruction mixture has less overall influence.
Figure 2.14: Typical speedup in percent (%) (in this case for the JACIT benchmark) ordered by increasing matrix size, from left to right.

pointers as a tree, a DAG or a cyclic graph [19]. Hwang and Saltz realized that it is of more importance how data structures are actually traversed instead of knowing the exact layout of a data structure. They integrated this idea in what they call traversal-pattern-sensitive shape analysis [27]. Integrating such an approach in our compiler could help in reducing the overhead introduced by the pool access tracing by traversing data structures autonomously in the
2.6 Conclusions

In this chapter, we presented and evaluated our restructuring compiler transformation chain for pointer-linked data structures in type-unsafe languages. Our transformation chain relies on run-time restructuring using run-time trace information, and we have shown that the potential gains of restructuring access to pointer-based data structures can be substantial.

Curial et al. mention that relying on traces for analysis is not acceptable for commercial compilers [12]. For static analysis, this may often be true. For
dynamic analysis, relying on tracing is not necessarily undesirable and we have shown that the overhead incurred by the tracing and restructuring of pointer-linked data structures is usually compensated for within a reasonable amount of time when data structures are used repetitively.

The restructuring framework described in this chapter opens up more optimization opportunities that we have not explored yet. For example, after data restructuring extra information on the data layout is available and could be exploited in order to apply techniques such as vectorization on code using pointer-linked data structures. This is a subject of future research.

Data structures that are stored on the heap contain object identifiers instead of full pointers. This makes the representation position independent, which provides new means to distribute data structures over disjoint memory spaces. Translation to full pointers would then be dependent on the memory pool location and the architecture. This position independence using object identifiers has been mentioned before by Lattner and Adve in the context of pointer compression [37]. However, with the pool restructuring presented in this chapter, a more detailed segmentation of the pools can be made and restructuring could be extended to a distributed pool restructuring framework.

The implementation presented in this chapter uses some run-time support functions to remap access to the proper locations for split pools. The use of object identifiers implies a translation step upon each load and store to the heap. These run-time functions are efficiently inlined by the LLVM compiler and have a negligible effect when applications are bounded by the memory system. The run-time support could in principle be implemented in hardware and this would reduce the run-time overhead considerably. We envision an implementation in which pools and their layout are exposed to the processor, such that address calculations can be performed transparently. Memory pools could then be treated similarly to virtual memory in which the processors also takes care of address calculations.

We believe the restructuring transformations for pointer-linked data structures that have been described in this chapter do not only enable data layout remapping, but also provide the basis for new techniques to enable parallelizing transformations on such data structures.