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 4
Pax C

The shape of pointer linked data structures is the result of two key aspects: control flow and input data. For example, a recursively typed structure such as the one shown in Figure 4.1, with more than one pointer\(^1\) in the structure may describe any kind of graph shape\(^2\). In the example, the type `Foo` may represent for example a binary tree, a sparse matrix node, a doubly linked list, a DAG or a generic graph. In fact, there is no way to infer what kind of shape the linked structure represents without taking the control-flow and, if necessary, the input data into account.

```c
struct Foo {
    struct Foo *a;
    struct Foo *b;
};
```

Figure 4.1: Struct with two recursively typed pointers. It can represent any kind of graph configuration.

While there have been many attempts to alleviate this problem. Most systems have either relied on low or high level approaches. The low level approaches deal with methods such as data remapping and pointer aliasing ([12, 36, 38, 47, 54]), and high level approaches deal with the shape of the

\(^1\)As C makes arrays and pointers identical, this is not entirely correct. But for now, let us consider pointers to be the same as references.

\(^2\)By treating multiple linked structs as being the same vertex in a graph, we can construct arbitrarily complex data structure shapes as long as the structs have two or more pointers. Single pointers can only link together straight chains, cycles and lassos.
structures such as whether a data structure is a tree, a DAG, or a generic digraph ([17, 19, 24, 26, 41]). However, the low level optimizations are not powerful enough to make use of specific data structure characteristics, and the high level approaches are either too costly to implement or require too much user involvement. Also the high level approaches lack the ability to provide runtime support to automatically perform analysis at runtime or to carry out runtime verification whether the higher level properties are violated.

Another issue is the more subjective quality of ease of programming. While high level languages may add advanced grammars to describe different data structures, these grammars have in many cases been too far abstracted from the normal programming language.

In this chapter we introduce a C programming language extension named Pax C\(^3\) which combines the efficiency of the low level approaches with the expressiveness of the high level approaches, while offering an easy model to program. At the same time, the additional code and attributes written by the programmer declares the intent of the pointer linked data structures to the compiler, and the conformance of the control flow to this intent can be verified at either compile or run time.

We have identified a number of properties that are important for the optimization of pointer linked structures. Firstly, coverage of linked data structures is a property that describes whether a specific path in a data structure visits all elements. This property enables foreach optimizations of pool allocated data structures. These optimizations serve two primary purposes, they ensure temporal locality and improved cache behavior, and they enable data parallelism within the pool.

Secondly, disjointness of linked data structures defines whether different paths are non overlapping or not. The property is closely related to aliasing, but we see it as a more high level property, where aliasing refers to individual pointers, and disjointness to the larger structure. For example, different branches in a tree are disjoint. Disjointness is useful for determining parallelism in the case where there are sequential dependencies.

Thirdly the direction of pointers determines whether two pointers are anti-parallel or if an object may be reachable through a combination of different pointers. The direction notion can improve aliasing analysis and enables the transformation of pointer chasing code into index increments (and for the anti-parallel pointers as decrements). This property is also usable for inferring disjointness in some cases. Unlike the coverage and disjointness properties,\footnote{Pax C was in the early stages an abbreviation for pointer axiom C, though it should not be considered to actually mean this anymore.}
it does not apply to traversal paths through the pointer structures but to pointers in the structure.

Fourthly we have the \textit{firmness} property. With firmness we are dealing with how robustly a structure follows the other properties over time. For example, during construction, a pointer based structure may not respect that two pointers are supposed to be anti-parallel, but after the construction the property will be valid. Firmness is a temporal quality, and is thus expressed in the code, while the other properties are spatial properties that are associated with the data types in the program. Firmness can for example be expressed as either static or dynamic pointer structures, where in the first place, the pointers building up the backbone in the linked structure are constant, and in the dynamic case the pointers may change to point at other objects within a covering set of object. The dynamic pointer structures should be seen as being more firm than structures that are modified by adding new objects allocated with for example \texttt{malloc}.

The language extensions described in this chapter allow the programmer to express \textit{directions}, \textit{coverage}, \textit{disjointness} and \textit{firmness} of pointer based structures. While the first three properties describe a data structure’s connectivity and shape properties, i.e. what does the structure look like, the \textit{firmness} property relates to how these properties change over time. For example a pointer linked structure may be constructed during start up of a program, while the program never changes the connectivity or link information; in this case the structure is stable after the initial setup and therefore firm.

While \textit{directions}, \textit{coverage} and \textit{disjointness} will be expressed using data type attributes (using the \texttt{__attribute__} syntax from the GCC and Clang compilers), the firmness property is a temporal property, expressed in the control flow using type qualifiers (similar to \texttt{const} and \texttt{volatile}) and pragmas.

This chapter is organized as follows: Section 4.1 looks into related approaches and their limitations, contrasting these to the Pax C extensions. Section 4.2 gives an overview of the Pax C extensions. Section 4.3 discusses the automatic detection of static pointer structures. Section 4.3.1 goes into detail about dynamic pointer structures. Section 4.4 discusses data restructuring mechanisms enabled by the extensions. Section 4.5 discusses the experiments used to test enabled optimizations. This is followed by Section 4.6 that list the results of the experiments. The experimental evaluation of the language extensions is followed by an analysis of the results in Section 4.7. Finally, conclusions are discussed in Section 4.8.
4.1 Limitations of Other Approaches

Low level approaches aiming to analyze and optimize pointer linked data structures, include for example Lattner and Adve’s Data Structure Analysis (DSA) [38], automatic pool allocation [36] and the structure splitting work done by others [12, 54]. Basically, in these approaches different properties of the connectivity of data structures and aliasing properties are used to optimize the data structures. The methods are based on compile time analysis, where the compiler determines whether or not different object are aliasing or not. The aliasing and connectivity information is used to optimize the data structures of the program such as automatic pool allocation and structure splitting, or to optimize the control flow when read after write dependencies can be resolved.

One of the main limitations of these approaches is that they are too conservative and that they miss optimization opportunities. Another limitation is that the aliasing relationships of more complex expressions can not be determined. As a result of too little (derivable) knowledge at compile time these methods must make conservative assumptions for further optimizations as more aggressive assumptions would result in errors for some cases. For example in the pointer analysis used by the DSA, data structures will be marked as disjoint only if they can be guaranteed to be disjoint. So, it may conservatively identify certain objects as maybe aliasing, while they should in-fact be treated as disjoint or as always aliasing. This is not to say that the DSA or other algorithms such as Stensgaard [47] are not good at what they do, but they are limited in what they can do as they are supposed to be a normal part of the compiler, and this means that they must not be too slow if the compiler is to provide a reasonable compile time.

DSA for example is context (but not flow) sensitive and able to identify disjoint data structures in a conservative manner. Conservative in this case means that structures marked as disjoint are guaranteed to be disjoint, but structures not identified as disjoint are not guaranteed to be fully connected (i.e. it does not exclude that a structure have disjoint regions). As such, the DSA is conservative albeit very powerful. Although the conservative approach of the DSA does offer the ability to implement automatic pool allocation as mentioned earlier, it is not possible to identify covering traversals of the pool and, for example, apply vectorization or other parallelization methods on the traversals. In fact, any sequential pointer chasing dependency still remain in the optimized data structure and code. The Pax C extensions solves this problem by allowing the programmer to declare covering traversals of data structures. These traversal patterns can then be matched with control flow which in turn allows the compiler to determine when a traversal visits all
elements in a linked data structure. Consequently, if the data structure is pool allocated it will parallelize the traversals, depending on whether there exists additional data dependencies.

The low level approaches are sometimes combined with runtime mechanisms. In [54] runtime tracing was used to optimize the order of the objects in a pool allocated pointer linked data structure. This method was limited as it was restricted to immutable pointer structures. The runtime tracing in the system had a very high overhead, so it was only used initially until the data structure had been optimized, after which the tracing mechanism was turned off. Consequently it was not possible to take into account dynamically modified pointers. Our extension, does on the other hand offer the opportunity to distinguish between static and dynamic pointer structures and for the latter part, allowing multiple optimization points for the same structure, where each optimization point may use different paths through the structure, or be conditionally executed based on programmer controlled conditions.

Among the high level approaches, several analyses and different language grammars have been developed. In Shape Types [17], Fradet and Metayer describe a context-free graph grammar allowing the expression of doubly linked and circular types, which is not possible using traditional type systems. They (like in this chapter) introduced an extension to the C programming language, and supported a kind of checking of the structures on initialization and modification (or reactions), based on a special syntax for the modification of the objects. The main limitation in their work was that the shapes where described using a context-free graph grammar that does not blend well with C. Furthermore, modifications of the structures required the use of a special tailored syntax (which allowed the compiler to prove that modifications were valid). So, while their system did extend C, the extensions where not directly compatible with existing C-code. Pax C on the other hand allows for the successive refinement of existing C-code, without any rewriting of the pointer code, and preprocessor directives can easily ensure that attributes are ignored and that the modified code still compiles using a standard C-compiler.

Other work carried out by Hendren et.al. [24, 26], known as ADDS and ASAP built on extensions to existing “struct” definition syntaxes. They introduced a sense of direction in pointer linked structures (not specifically as a C-extension, but it was aimed at C-like languages), their initial work introduced a way to add directional quantifiers to existing record types. They later extended their system with a more general form of aliasing definitions and descriptions. While being effective in determining aliasing properties of different pointer chains, their approach did suffer from the fact that, as, with the DSA and automatic pool allocation, it is not possible to define coverage of
pointer linked structures. This is further illustrated by Figure 4.2, where one row is not directly reachable from the array of row pointers. ASAP and ADDS did allow for the distinction between the vertical and horizontal dimensions of such structures, but was not able to describe the absence of the hidden nodes.

In addition to the mentioned language extensions, there has been other high level work such as shape analysis [19]. In shape analysis the compiler analyze the actual shape of a data structure based on for example pointer assignments and allocations. Depending on the system, the resulting shape estimates may differ in both accuracy and correctness. The system was able to reasonably accurately detect the more specific types of structures. While the tree detection could reasonably easily be used for data coverage and disjointness analysis by analyzing the traversal patterns in the code, the more general types of structures detected are difficult to use for this purpose, except for the limited cases of local alias analysis. The disjointness properties introduced by Pax C are stronger than this and can describe global aliasing patterns in a larger structure, based on access patterns with conditions. These disjointness patterns even work on cyclic structures and DAGs.

Another high level approach named PALE [41] was presented by Møller et.al. PALE defined a formal language to describe data structure invariants that could be checked at compile time against a restricted imperative language. The main problem with PALE consist of the fact that execution time and memory requirements of the proofs were too large for practical use. For
example the verification of a red black tree insert took 35 seconds and used 44 MB of memory to check 57 lines of code. While this was the worst test case they provided, the mean and medians both ended up at roughly 30 lines of code, 4 seconds and 8 MB of memory. These results indicate that the computational and space requirements of their methods prohibit a practical application to real world programs that typically consists of millions of lines of code. Even on modern machines, as processor performance has improved considerably since the PALE work was published, the execution time would end up in hours if linearly extrapolated for a 10 million line large program (this is a fair assumption assuming that a program consist of properly encapsulated data structures, all who utilize the PALE logics). In Pax C, we assume a limitation on what can be derived, i.e. more conservative reasoning, and do not let the compiler handle everything. Instead, the attributes in Pax C have been carefully vetted to be verifiable at run-time, and runtime-checks can be emitted if needed.

Note that some systems simply assume that structures are correctly linked. So, a programming error or an invalid object, loaded from a file could potentially break the invariants for the data structure. In the case of PALE this would not matter as it is based on formal proof. However, for Pax C, and also for the ADDS and ASAP, the properties can be violated during runtime. With ADDS and ASAP if the additional data structure properties are not respected by the code, the compiler could in principle generate invalid optimizations that would be very hard to track down. The approach in Pax C is simple, let the program be verified at runtime using a relatively simple model that can catch as many violations as possible when a data structure is used incorrectly.

4.2 Pax C Extensions

Pax C is a set of C extensions forming a strict superset of C, that allow for a more sophisticated description of recursive data structures. Many of the attributes are applied on pointer fields in struct definitions (or on the struct definition itself). There are however a few extensions that are used within function bodies to help the compiler optimize the code. This approach is in stark contrast to the use of other C-extensions that mostly consist of directives embedded in the control flow e.g. OpenMP and Cilk [5].

The first extension of Pax C is the annotation of pointers in data structure types with attributes or axioms that describe either local or global pointer properties. Local pointer properties are the pointer properties that apply to one object and possibly also the object’s neighbors; global pointer properties
are those that apply on multiple objects within a data structure. The global properties make use of traversal paths in most cases, these paths describe how a structure is traversed using a regular expression like syntax. The second extension consists of the static-pointer structure type qualifier. Essentially, a static pointer structure pointer identifies an object where the pointers in that object and in all objects reachable from it are constant. As such, it is known that the shape of the object graph will not change further on in the program. The programmer can define the point from which the data structure’s pointers will no longer change in the program by manually adding a type qualifier (similar to the existing C type qualifiers like restrict, volatile and const) and casting a pointer to a non static-pointer structure into a static-pointer copy of that structure. At this point, the casting operation will trigger a deep copy and restructuring of the data structure, where the traversal pattern is determined from the pointer attributes. In addition to the static-pointer qualifier Pax C introduces the notion of dynamic-pointer structures. These dynamic pointer structures are similar to the static counterparts, but they allow for the repointing of the recursively typed pointers to other already allocated objects.

4.2.1 Conditional Traversal Patterns

The Pax C extensions consist of a number of pointer attributes, type qualifiers and pragmas. The pragmas and the pointer attributes are used together with conditional traversal patterns (CTP). These patterns describe how to traverse a structure.

A pattern consist of a sequence of field dereferences separated by the dot operator (“.”). Between the dot operators, the fields may be expressed as either, field names or using other expressions, such as the | operator that follows each side of the | and the trailing expression (before the next branch). The branch operator is related to the array operator [ ], that does the same thing, but with arrays instead of named fields. The * operator works in similar ways, but expands the expression it is trailing, so it is followed until it reaches null or the starting point when expanding a cyclic field. Note that the * operator only follows single fields or conditional single fields.

The condition operator allows the selection of a path based on a conditional variable/expression, local to the last object in the chain, or a C variable in the case of the pragmas. The conditions are written \{cond\?a : b\}, where a is used if cond is true and b if it is false. The conditions normally refer to fields in the current object. E.g. foo.{bar == 0?left : right} expresses the traversal:
4.2. **PAX C EXTENSIONS**

```c
current = current->foo;
if (current->bar == 0) {
    current = current->left;
} else {
    current = current->right;
}
```

Finally, the comma operator traverses the full expression to the left of the comma and then the one to the right of the comma. This allows for the restart of the traversal patterns at controlled points.

The expressions are easy to turn into C-code and to understand. The template for conditional traversals has already been shown. For the looping constructs, like `*` and `[ ]`, the templates are also relatively simple. An array traversal can be emitted using the following code:

```c
array = current;
for (int i = 0 ; i < arraylen(array) ; i++) {
    current = array[i];
    // Rest of expr
}
```

Note that introduced variables (e.g. array in this case) should be uniqued for the traversal. This can be done by appending a serial number to the name, or by embedding it in a scope.

The `*` works in similar ways, except that the loop is a while loop on the fields. A star traversal can be emitted using the following code:

```c
while (current) {
    temp = current;
    // Rest of expr
    current = temp;
    // Traversal (e.g. current = current->next, if
    // the expression is next*)
}
```

For the traversal expressions discussed here, we can insert the visited objects (pre-order) in a sequence, that means that backtracking nodes are excluded from the sequence since they have already been embedded in it and that the successor of a the leaf node in a branched traversal will not be the parent node of the leaf, nor will it be the branching node, but the next unvisited node starting from the branching point. This sequence we will denote
typedef struct S {
  size_t bar_len;
  Foo *bar __attribute__((length(bar_len )));
  Foo *baz __attribute__((single ));
} S;

Figure 4.3: Example of length and single attributes

\( T_{seq} \) in the discussion.

In the following section we will give definitions of the attributes that Pax C supports.

### 4.2.2 Single and Length

The *single* and *length* attributes were defined to work around issues in the C-programming language. In C, pointers to single objects (i.e. references) cannot be distinguished from arrays of objects, and pointers to arrays are not associated with the length of an array. The *single*-attribute forces a pointer to be a reference to one and only one object, while the *length*-attribute allows the specification of an expression that can derive the length, based on data available at the definition point. For pointers embedded in structs, this can refer to a field in the struct. Example of the use of these two attributes are given in Figure 4.3.

Note that these attributes simply associate available length information to the arrays in question. The array length is needed in order to use the array expression for iterating (but not for accessing single objects).

The attributes adds missing information to the C-programming language. Consequently, the property can be used in order to verify during runtime that array accesses do not go out of bounds and that directly assigned blocks from malloc and the related allocation functions are of the assigned sizes. Standard optimizations to bounds checking used by programming languages such as Ada and Java apply to these checks.

### 4.2.3 Acyclic and Cyclic

The *acyclic* and *cyclic* attributes provide known termination points for common recursive pointer patterns. In effect, as a computer has finite memory, there are three possible shapes for structures linked using one given pointer field: *full cycles*, *straight chains* and *lassos* (see Figure 4.4). Under normal
4.2. PAX C EXTENSIONS

Figure 4.4: Lasso Graph

```
typedef struct Foo {
    struct Foo *bar _attribute__((acyclic));
    struct Foo *baz _attribute__((cyclic));
} Foo;
```

Figure 4.5: Example of acyclic and cyclic attributes

circumstances, the compiler must assume the worst and that a chain may be in any of these formats.

For full cycles and straight chains the termination points for a full traversal of the structure (when \( p->\text{next} == \text{start} \) and \( p->\text{next} == \text{NULL} \) respectively) are easy to derive. However, for lasso shapes this is more difficult.

An example of the usage of the acyclic and cyclic attributes is given in Figure 4.5. In the example, the attributes guarantees that, for all pointers \( p \), to objects of type \( \text{Foo} \), firstly, the traversal \( \text{while (p) p = p->\text{bar};} \) will terminate as \( \text{bar} \) is acyclic and therefore must end with a NULL pointer, and secondly, the traversal \( \text{tmp = p; while (p->\text{baz} != tmp) p = p->\text{baz};} \) will terminate since \( \text{baz} \) is acyclic and the loop terminates at the starting node.

Statements like these can be verified. The violations for the attributes occur if the pointer is acyclic and a full cycle or lasso shape exist, or if the pointer is cyclic and the chain represents a lasso or a NULL terminated chain.

The detection of cycles is a well known problem. One of the more commonly used methods traverses a potential cycle and samples the visited node at increasing intervals. Normally, the distance between the samples increases by a power 2 for every taken sample. This way, cyclic chains will always be detected within two iterations, even if the chain is a lasso. Consequently, iterations along cyclic or acyclic pointers can be instrumented with these checks. If the runtime checks find a cycle before it reaches the starting point of a cyclic pointer, or before it reaches NULL on an acyclic pointer, the program can raise an error. There are ways to avoid these runtime checks if certain conditions hold, though.

Both properties can in some cases be proven by induction. The starting condition is that in the acyclic case, the acyclic field is initialized to NULL,
typedef struct Foo {
    struct Foo *bar __attribute__((inverse(baz)));
    struct Foo *baz __attribute__((inverse(bar)));
} Foo;

Figure 4.6: Example of Inverse Attributes Use

and in the cyclic case, the field is initialized to point at the same structure it is part of. The compiler can check for this and ensure that after calls to allocation functions (e.g. malloc), the fields are set accordingly before the object is used for anything.

For inserts of individual objects nothing needs to be checked, as long as the initial condition holds for the insertion point and the inserted object. The resulting chain is also null terminated or cyclic.

However, insertion of chain slices is a more complex issue. While, this can simplified in certain condition, it is very complex to do so as the inserted slice may or may not be a single chain, consequently it must be verified that the inserted chain slice is not actually a slice of the destination chain, and this is not always possible. In this case, when not inserting single objects, the checks will be deferred until restructuring time.

### 4.2.4 Inverse

A pointer field $a$ in an object $p$ is the inverse (or antiparallel) to another pointer $b$ if after following $a$, $b$ points back at $p$. The property asserts that as long as $p$ and $p->a$ are not NULL, then $p->a->b == p$. Common examples include binary linked lists and trees with parent pointers.

Inverse pointers are of interest in order to assist in the verification of cyclicality, and in order to allow for more advanced restructuring opportunities.

An example of the usage of the inverse attribute is given in Figure 4.6. The usage of the attribute in this Figure implies that for all pointers $p$ to objects of type $Foo$: $p->bar->baz == p$. This attribute is assumed to hold at all times, except during transient modifications.

If a code segment modifies an inverse pointer in object $p$, the inverse must be modified as well to point back to object $p$ before the fields in question are read and used for something eles. This can be checked statically per function. However, if a function does not uphold the attribute, the compiler may emit a warning and make a note that the function made the object $p$ invalid and then proceed with emitting checks at runtime.
4.2. PAX C EXTENSIONS

```c
typedef struct Foo {
  struct Foo *x _attribute_((cyclic)); // Or acyclic
} Foo;

typedef struct Bar {
  struct Foo *a _attribute_((first(x)));
  struct Foo *b _attribute_((last(x)));
} Bar;
```

Figure 4.7: Example of First and Last Attributes

4.2.5 First and Last

The `first` and `last` properties are used to define the starting and ending points of a pointer chain. Depending on whether the chain follows an acyclic or a cyclic pointer field, the attributes have slightly different meanings. By knowing the start and the end of a pointer chain, it is possible to infer that iterations between the start and the end (or the end and the start following inverse pointer) will visit the entire chain. In the cases where the pointer chains can be serialized, this means that it is trivial to determine that the iteration visits every element in the chain. In addition to this, the attributes can help with the verification of the acyclic and cyclic attributes.

Figure 4.7 illustrate the usage of the attribute. In the acyclic case, for all pointers `p`, pointing out objects of type `Bar`, the attribute `first` implies that when following the field `x` for all objects `q` of type `Foo`, there exists no `q` where `q->x == p->a`. For the `last` attribute on the other hand, if the field `b` of the type `Bar` has the last attribute, then for all objects `p` of type `Bar` `p->b->x == NULL`.

In the cyclic case this is a bit different, since there are no logical starting points of a cyclic chain: the first and last attributes are defined with respect to each other. If the pointer `p` is first, and `q` is last, with respect to the field `x`, then `q->a == p`.

For the cyclic case, the verification is obviously trivial at runtime. However, in the acyclic case, it is only easy to verify in the case that the relevant field has an inverse. In this case, if the pointer `p` has the attribute first with respect to the field `a`, and field `b` is the inverse of `a`, then `p->b` must be `NULL`. For the case where there is no inverse, the compiler is expected to issue a warning about it. Note that violations can be always be detected using brute force approaches while restructuring, where we could tag visited objects, and which field reached the object.
typedef struct Foo {
  struct Foo *parent;
  struct Foo *sibling
    __attribute__((ident(sibling->parent)));
  struct Foo *child;
} Foo;

Figure 4.8: Example of Ident Attribute

4.2.6 Ident

In some cases, different pointers point to the same object in a predictable way. For these cases, the ident attribute exists. It can for example be used to infer tree structures in the case where a parent only has a pointer to one of the children who in turn have pointers to its siblings.

The ident attribute uses the conditional traversal patterns to identify which objects are identical. As paths cross many objects, the attribute cannot refer to all the visited objects, instead it refers to the leaf points in the traversal paths. An example of the ident attribute is given in Figure 4.8. In the figure, the sibling nodes parents will all point out the same parent.

4.2.7 Covering and Disjoint

The covering and disjoint attributes utilize the conditional access patterns in order to describe whether or not the traversals will visit all objects or whether the traversal will be completely disjoint.

A conditional access pattern will traverse multiple objects and the sequence of objects, known as $T_{seq}$, may visit objects several times in some cases (e.g. in case there are multiple paths to the same object).

For the covering attribute, the following is assumed to hold: for all objects in $T_{seq}$, there are no paths from any of these objects to an object of the same type, unless those objects exist in $T_{seq}$. Note that, in some cases, there are several pointers, all pointing to different covering sets (e.g. two different sparse matrices), for this reason the attribute has a second argument, a set id. Thus, the attribute applies only to the objects identified with the set id.

For the disjoint attribute, the definition is simpler: for a disjoint CTP the resulting $T_{seq}$ is a set, or simply, every object in $T_{seq}$ occurs only once.

Verification of disjoint and covering is deferred until restructuring. Where it is reasonably trivial. If we restructure using some path. The path can be checked reasonably easily by tagging visited objects, if the path matches the
4.2. PAX C EXTENSIONS

typedef struct Foo {
    struct Foo *x __attribute__((acyclic));
} Foo;

typedef struct Bar {
    struct Foo a[] __attribute__((covering(a[].x*, 0)))
       __attribute__((disjoint(a[].x*)));
} Bar;

Figure 4.9: Example of Covering and Disjoint Attributes Usage

covering or disjoint paths. Traversals along disjoint paths may never reach
an already tagged object, and traversals along covering paths may never visit
objects that have pointers to objects not in the path. As will be seen in Sec-
tion 4.4 the restructuring operations, must rewrite all the pointers to moved
objects. The remapping is done in a second pass after the objects have been
moved. In this case, if a pointer is not found to be remapped to a new loca-
tion, then the sequence cannot be stemming from a covering path. However,
the operation may be costly, so like with most runtime checks, it should be
controllable by compiler switches.

4.2.8 Static Pointer Structures

Pax C makes use of a notion we call static-pointer structures that can be ap-
plied on certain data structures. In essence, static-pointer structures have con-
stant pointers. The formal definition is a bit more complicated: For a static-
pointer structure object, the object is used in a type safe manner; all pointers
in that object that point to aggregate objects that in turn contain pointers, are
immutable and point at static-pointer structure objects.

It is arguably very difficult for a compiler to determine which objects are
static-pointer structures and where in the programs they are so. In practice it
is also impossible to do so in many cases, especially when the compiler has an
incomplete view of the program. Therefore we introduce a type attribute that
can be attached to structures in the same way the structure can be declared
current, volatile or restricted. It is up to the programmer to add casts at
relevant points in the code that declare to the compiler that a data structure
is a static-pointer structure. This information can be used by the compiler to
generate data structure transformations.

As mentioned in the introduction, the automatic detection of static-pointer
Figure 4.10: Static-Pointer Casting

structures is conservative in nature. The automatic detection will help the programmer to incrementally add attributes to the code. A method able to detect static-pointer structures is described in Section 4.3.

4.3 Conservative Static Pointer Detection

In some cases, it may not be trivial for the programmer to determine that a data structure is a static-pointer structure. In the case that the programmer has only been setting the pointer attributes in the data structures, it is possible for the compiler to detect naturally occurring static-pointer structures. In order to do this, it is possible to employ an extended version of the DSA (the full details of the DSA being out of the scope of this chapter, but it is briefly described in Chapters 1 and 2). The modifications add per-field flags and an additional top-down pass, neither of which should be resulting in any major performance hits due to their low complexity.

The DSA employs a local analysis, followed by a bottom up and top down analysis of the call graph, where local analysis information is passed upwards (i.e. due to functions returning values or modifying objects passed into the functions), and merged (when a pointer variable has been used or assigned in multiple branches) and then passed downwards in the call tree.

Note that the local analysis followed by bottom up and top down passes can easily be used to propagate information about verified attributes. When nodes are merged (because they are defined in different branches), the validity flag for the attribute is simply “bitwise anded” together, meaning that if one branch results in an invalid attribute then the merged representation is invalid as well. This is done after the local analysis has determined whether the attributes are valid, assuming that the preconditions are valid.

In order to detect static pointers, one need to enable per-field mod / ref flags, allowing the compiler to detect in which contexts the pointers within a data structure are written to. The second addition is to add two additional per-field flags. These are the free and the nullification flags. The free flag is set if a field escapes to the free function and the nullification flag is set if a pointer field is written the constant value NULL (in this case, the mod flag is not set). When this is done and the local phase of the DSA has been completed we do a
4.3. **CONSERVATIVE STATIC POINTER DETECTION**

![Diagram of static-pointer structure detection]

Figure 4.11: Static-Pointer Structure Detection. Flags are: $R =$ read, $N =$ nullify, $F =$ free.

bottom-up type merge (where all DSA node types are merged) and a top-down traversal (where the same is done). It is then possible to do another top-down traversal (starting with the main function). During this traversal, we build a per-data structure inter-procedural dependence graph where the nodes that create the data structure will be dominating the successors (i.e. they will be executed earlier in the program flow). This step is context sensitive like the DSA and not flow sensitive\(^4\). In this dependence graph we are looking for a pattern where the first flags are modifying the pointers and the rest of the graph is not modifying the pointers (except for the last couple of nodes where the pointers may be nullified). The pattern that is searched for is illustrated in Figure 4.11.

After detecting a **static-pointer structure**, the compiler can insert a **static-pointer cast** just before the **static-pointer region** (see the middle **usage** node in Figure 4.11). This cast is inserted just before the usage phase. For the tear-down phase, the original data structure (not the converted one) should be passed along, this is safe since the structure in question is no longer used. At this point it is useful to insert a runtime call to release the **static-pointer structure**.

### 4.3.1 Dynamic-Pointer Structures

In contrast to a **static-pointer structure**, a **dynamic-pointer structure** allow the pointers to be changed and objects relinked, however it does not allow for new objects to be allocated and inserted in the pointer linked structures (the programmer can overcome this limitation by using fixed sized pools of objects). Casting a normal pointer to a **dynamic-pointer structure** will otherwise work similar to the **static-pointer structures**.

The dynamic-pointer structure was added to the Pax C extensions in order

\(^4\)For a summary of the terms context and flow sensitive, see [25]
to be able to handle problems where an iterative solver is modifying the pointer structures slowly converging on the final result. Since, the dynamic-pointer structures allow the pointers to be modified they are more flexible for the programmer than the static-pointer structures defined in the previous section. On the other hand, they do not allow for certain optimizations that rely on having a known iteration space.

4.4 Restructuring

When an object is casted into a static-pointer copy, the compiler is free to reorganize and restructure the pointers in the data structure. Although, some restructuring can possibly be done before this (such as some forms of pointer compression, and other type modifications), the static-pointer property does allow for more options.

A type-specific automatic restructuring function can be generated by the compiler when it knows the attributes mentioned in section 4.2 and calls to this restructuring function at the locations where a normal object is casted into a static-pointer copy. The main premise for generating such a function is to be able to determine a set of covering (and preferably disjoint) paths in the data structure.

Naively, it is always possible to determine a covering set by a DFS traversal of the data structure. However, this is not practical for several reasons. For example, a DFS will in many cases not correspond to the natural iteration order of a more complex data structure and it will have to deal with a lot of bookkeeping information to avoid endless recursion in the presence of cycles.

The rewrite operation does two things, firstly it transforms the pointer types, and secondly it does a deep copy of the rewritten data structure (following the covering path) into a new single data block (an array). The pointers are converted in order to refer other objects within the array using indices. Consequently, when converting a pointer-linked data structure into the static-pointer version, there are three forms of pointers to consider: indices within the new data block, fat pointers that refer to another data block and slices that refer to a subset of objects within a data block. An index is used when there is a self recursive pointer within the same set. A fat pointer, which is a tuple of an index and an array pointer, is used as pointers to single objects in some other set (or data block). A slice is a triple of a start index, end index and an array pointer. Slices are used when pointing out the first object in a known subset (if this is in the same array as the current object, it can be shortened into just the start and the end index), for example, when we point
4.4. RESTRUCTURING

<table>
<thead>
<tr>
<th>Original Type</th>
<th>Introduced Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>struct A { A *a; B *b; }</td>
<td>struct A_1 { int a; B_fatptr b; }</td>
</tr>
<tr>
<td></td>
<td>struct A_array { int len; A_1 elems[]; }</td>
</tr>
<tr>
<td></td>
<td>struct A_fatptr { int idx; A_array *arr; }</td>
</tr>
<tr>
<td></td>
<td>struct A_slice { int start; int length; A_array *arr; }</td>
</tr>
</tbody>
</table>

Table 4.1: Type Conversion. The o is used as a generic operator, e.g. == or !=.

<table>
<thead>
<tr>
<th>Original</th>
<th>Translated</th>
</tr>
</thead>
<tbody>
<tr>
<td>*p</td>
<td>p.arr-&gt;elems[p.idx]</td>
</tr>
<tr>
<td>p=NULL</td>
<td>p.idx=0</td>
</tr>
<tr>
<td>q=p-&gt;R</td>
<td>q.idx=p.arr-&gt;elems[p.idx].R</td>
</tr>
<tr>
<td></td>
<td>q.arr = p.arr</td>
</tr>
</tbody>
</table>

Table 4.2: Statement Conversion

out the first object in a list like structure such as a row in a sparse matrix. The introduced types are illustrated in Figure 4.1.

Since the types are modified, the addressing code needs to be rewritten. Essentially all pointers to static-pointer objects will be turned into fat pointers. This includes pointers passed as arguments to functions. Initially only three translations need to be done in the code when accessing the fat pointers. We show this in C syntax in Figure 4.2.

This will not necessarily result in any improvements of the performance of the code in question. Several additional optimizations are performed on top of this. Most important is the rewriting of pointer-chasing loops into index-incremented loops (see Figure 4.3). This is possible if the pointer chasing goes in the same direction as the covering and disjoint paths and the traversal covers a full path or the full range between two known elements, such as a full structure or a slice.
while (p) { p = p->R; } for (; p.idx < p.arr->len ; p.idx ++)

Table 4.3: While Loop Conversion

Note that depending on whether or not the initial data structure has been pool allocated or not, the rewriting mechanism can either use an indirection array or a hash-table to store the old to new pointer mapping. The rewrite is a two pass affair, where in the first pass the data structure is copied over to the new data block, and in the second, the pointers are rewritten to point at the new data locations. The only pointers that can be updated on the first pass are those that point at the “next” object.

4.5 Experiments

4.5.1 Sparse Lib

The SPARSE library (see [32, 33]) utilizes orthogonally linked lists to represent sparse matrices. The library provides a matrix frame type describing the matrix structure (i.e. the size, starting points of rows and columns) and pointer-linked matrix elements that encapsulate a non-zero element. The data structures are well suited for optimization using the Pax C extensions. The updated types in the (not showing irrelevant data fields) is listed in the following code:
4.5. EXPERIMENTS

```c
struct MatrixElement {
    double Real;
    struct MatrixElement *NextInRow
        __attribute__((single))
        __attribute__((acyclic));
    int Col;
    int Row;
}

struct MatrixElement *NextInCol
    __attribute__((single))
    __attribute__((acyclic));

struct MatrixFrame {
    int NrOfElements __attribute__((set.size(1)));
    struct MatrixElement **FirstInCol
        __attribute__((covering("\[*].NextInCols", 1)))
        __attribute__((disjoint("\[*].NextInCols")));
    struct MatrixElement **FirstInRow
        __attribute__((length(Size)))
        __attribute__((covering("\[*].NextInRow", 1)))
        __attribute__((disjoint("\[*].NextInRow")));
    struct MatrixElement **Diag
        __attribute__((length(Size)))
        __attribute__((in.set(1)));
    int Size;
}
```

In the code listed above, the covering attributes not only specify the paths through the matrix, but also the set identifier. The set identifier is needed in case the matrix frame would point out two different disjoint matrices. The definition of disjoint rows and columns allow the compiler to not only restructure the data, but also to expose the inherent parallelism in a loop over the rows. In the matrix type, it is not possible to infer parallelism automatically in the general case as the code may modify the matrix contents during iterations.

We carried out experiments using a number of kernels based on the SPARSE library data types, namely `dsolve`, `jacit`, `pcg`, `spmatmat` and `spmatvec`. These kernels are part of the SPARK00 [51] benchmark suite. Most of the kernels traverses the matrices in either row or column order. There is one notable exception, `dsolve`, that includes the traversal of an LU-factorized matrix. In this case the matrix is traversed partially in row-wise order and partly in column-wise order. Figure 4.12 shows the average speedup of the different kernels when the matrix data is read using different memory orders. For row-wise ordered data, the restructuring is not helping. This is due to the fact that matrix is already perfectly ordered in memory. For the column-wise order, the data is not ordered in memory in the same way as the traversals, therefore we can see an improvement in performance here. For randomly ordered data, the data ends up being well ordered after the restructuring, so the improvement

---

5 Naturally, a loop not modifying the matrix contents can have its row or column traversals executed in parallel.
Figure 4.12: SPARSE lib kernels speedup factors grouped by benchmark.

is considerable.

4.5.2 MCF

MCF is known as 181.mcf in the SPEC2000 (or 429.mcf in SPEC2006) benchmarks. Previous work where MCF has been targeted for optimization includes among others [20] where the MCF data structures were subjected to peeling, splitting, and field reordering. These transformations are not global in the sense that they consider the connectivity of the objects, rather than transform individual objects into a different layout.

MCF is implemented using a structure containing an array of nodes and an array of arcs (each node also has a number of implicit arcs relating to the spanning tree that is being computed). The main data structure is traversed multiple times with different (potentially) conflicting access patterns. For example, the function update_potential traverses the nodes following the spanning tree using a DFS pattern, however, other functions iterate over the node array and traverses chains of nodes if certain conditions are met. In fact, reordering the nodes after the DFS traversal will slow down MCF in total (although the update_potential function itself runs substantially faster).

In our experiments we show that we have been able to achieve restructuring in MCF without the use of tracing, on which the restructuring systems to this day have been relying on (e.g. [54]). MCF is an iterative solver, as such the
As is well known, to determine how to optimize a program one needs to know the hotspots where the program is spending most of its time. We used profiling to determine that in MCF, the execution time was primarily centered around three functions. These functions are `refresh_potential`, `price_out_impl` and `primal_bea_mpp`. For the reference data set, the functions were using 96% of the program’s execution time. The simplified call graph in figure 4.13 illustrates this further. As can be seen, any optimization for the code must target these functions or seriously suffer from diminishing returns.

Obviously our first attempt was to optimize the access pattern for the `refresh_potential` function that was the most costly of the three. The optimization centered around the fact that MCF only modifies a small part of the data structure as the program gradually tries to build up the solution. So, essentially by adding the restructuring pragma to the beginning of the `refresh potential` function, using the access pattern used in the function, it was possible to carry out the restructuring step.

Essentially, the `refresh potential` access pattern is a DFS. This DFS is described by the attributes that can be set on the MCF data structures. By stat-
ing that there is a covering and disjoint path in the \texttt{network.t} structure type using the following expression: \texttt{covering(nodes[0].(child|sibling)*)}, The permutation vector is then generated following this traversal and the nodes are reordered.

This optimization however caused slowdowns in the other hotspots and although the \texttt{refresh\_potential} function ran substantially faster (40\% including overhead from the data restructuring), the new access pattern slowed down the program as a whole.

As the \texttt{refresh\_potential} function did result in slowdowns, the other hotspots had to be reconsidered. The \texttt{primal\_bea\_mpp} function was determined to use a much too complicated access pattern for both manual and automatic analysis. Instead the \texttt{price\_out\_impl} function was considered for optimization. The function essentially checks every third arc and if a tag value is not set to a constant value named FIXED, it traverses the nodes for the arc backwards (i.e. from the tail and backwards). The access pattern can be codified in the following way: \{\texttt{net.arcs[i=1..$length, step 3].ident != FIXED}\} \texttt{? net.arcs[i].(tail.mark)*}. By using the pattern given here, we were able to speed up MCF as a whole compared to the non (access-pattern) optimized version. The steps needed are to ensure that the \texttt{length} and \texttt{covering} properties of the network type are known, then the restructuring pragma needs to be added to the code.

\begin{verbatim}
typedef struct {
    node_t *nodes
        __attribute__((length(stop_nodes-nodes)))
        __attribute__((covering(nodes[0 .. $length])))
        __attribute__((covering(nodes[0].(child|sibling)*)))
        __attribute__((disjoint(nodes[0].(child|sibling)*)))
    node_t *stop_nodes;
    arc_t *arcs __attribute__((length(stop_arcs-arcs)));
    arc_t *stop_arcs;
} network_t;

#pragma restruct (node_t) net using\%
    {net.arcs[i=1..$length, step 3].ident != FIXED}\ ?\%
    net.arcs[i].(tail.mark)*
\end{verbatim}
4.5. EXPERIMENTS

Note that the mark field is not an arc pointer but a pointer sized integer. However, the field is used as a pointer in the code using type-unsafe casting and the benchmark had to be modified by changing the type of the mark field to the proper pointer type. In the future, Pax C may be extended with casting operations in the CTP syntax.

4.5.3 Parallelizing Refresh Potential

*Refresh potential* traverses the entire spanning tree of the graph of nodes. The only intra node dependencies that exist are from the parent nodes. As such, the function is suitable for parallelization using the traditional *divide and conquer* approach. A complication for this is that the trees are not balanced, meaning that additional steps need to be taken to properly parallelize the function.

We show how parallel execution of the refresh potential function can be implemented with minor changes to the code. The first step is to assign the covering and disjoint attributes to the MCF data structure as described previously. The second step is to rewrite parts of the *refresh_potential* function using a recursive traversal instead of the current pointer chasing one. When testing this with the large reference data set from MCF, we saw that a recursive version did not result in any performance differences compared to the standard iterative traversal. In this case, rewriting the code using recursion makes it easier to analyze the function.

The function can then be parallelized by invoking some of the recursive calls in parallel and waiting for the call to finish before returning from the caller. This is similar to how the *cilk* [5] programming language handles parallelism. Our manual attempts with the *cilk* language, however, did not succeed due to the heavy overhead. On the other hand the *cilk* version served as a conceptual proof. A two thread *pthreads* based solution was successful in speeding up the code.

Essentially, the core part of the refresh potential function is rewritten as illustrated in Figure 4.14. As seen, the *disjoint* attribute is embedded in the *node_t* type. Without this addition, the compiler would have to assume that the subtrees were fully aliased and would not be able to proceed with the parallelization attempts.

The compiler now has to analyze the recursive calls of the function, these are in the direction of the sibling and the child pointers. Each function also accesses the predecessor. Table 4.4 illustrates the dependencies for the refresh potential function extended one step. The simplified dependencies show the actual dependency after taking into account the *ident* and *inverse* attributes. The calls to *refresh_pot* have the traversal patterns: a) sibling.(sibling|child)*
typedef struct node_t {
    struct node_t *predecessor
        __attribute__((single))
        __attribute__((ident(sibling.*.predecessor)));
    struct node_t *child
        __attribute__((single))
        __attribute__((inverse(predecessor)));
    struct node_t *sibling
        __attribute__((single));
} node_t
    __attribute__((disjoint(
    predecessor.(predecessor|sibling)*,
    ((child|sibling)*)))
    __attribute__((disjoint(
    (child.(sibling|child)*),
    (sibling.(sibling|child)*))));

int refresh_pot(node_t *node) {
    int a, b = 0, c;
    if (!node) return 0;

    a = refresh_pot(node->sibling);
    if (node->orientation == UP) {
        node->potential = node->basic_arc->cost
        + node->pred->potential;
    } else {
        node->potential = node->pred->potential
        - node->basic_arc->cost;
        b ++;
    }
    c = refresh_pot(node->child);
    return a + b + c;
}

Figure 4.14: Rewritten Refresh Potential
and b) \textit{child.(sibling|child)}$. If the two patterns are independent (i.e. disjoint), and do not access other data except from reading, then clearly the function calls can be called in parallel. In this case these patterns match the programmer-specified attributes exactly. The traversals are thus disjoint. The additional access in the function using the predecessor node is also clearly not interfering with the traversals, which can be seen from the other disjoint attribute. The only direct dependency is the \textit{child} node’s potential field which depends on the current node’s potential field. Therefore, it is possible to conclude that the first recursive call on the sibling may be executed in parallel with the remainder of the function.

When it is known that the traversals are disjoint, the compiler needs to figure out how often these parallel calls should be done. Preferably, all the calls taking the \textit{sibling} pointer would be parallel. Unfortunately this does in most environments generate a substantial overhead. Our solution to this was to utilize the mechanisms in the restructuring system and add a periodic count of the subtree sizes for each node. The function was then cloned and the entry point modified to check whether the subtrees (reached by traversing the child and the sibling pointers) were of sufficient sizes. If both subtrees are of different sizes, the sibling node is handed away to a worker thread which the current thread waits for before using the result from the function call.

Note that only one branching point was used. The main drawback of this is that all parallelism will not be exploited, however, as will be shown later in the experimental results, it still yielded decent results.

### 4.6 Results

Our experiments were carried out on a number of different machines with somewhat varying results. Three major experiments were carried out during the exploration of the MCF code, Firstly, we investigated optimization of the \texttt{refresh\_potential} function only. Secondly, we investigated an optimization focusing on the \texttt{price\_out\_impl} function. The code was passed through the -
O3 optimization flag of the clang compiler to ensure that we were not repeating standard built in optimizations from the compiler. Thirdly, we focused on parallelization speedups and the refresh.potential function was parallelized with a single branching point taking into account the subtree sizes for the sibling and the child pointers.

The refresh potential test was executed by comparing a base version without restructuring, with variants that trigger restructuring every nth call to the function. We tested several intervals for the restructuring. Another test also compared two different versions, one generic using a very fast hash table for mapping old pointers to new pointer values (which is needed when objects are not allocated in arrays) and a second version using an index based remapping vector. The index based remapping vector is applicable on many different data structures, and should be used whenever possible. For this experiment we used a small data set\textsuperscript{6}.

The price out optimizations were tested using a hybrid access pattern method. Essentially two access patterns were used, the first one being the covering pattern from refresh potential and the pointer attributes; the second one being the pattern used in the price out impl function. The two access patterns were interleaved and tested with different combinations (for example 2 nodes using one pattern followed by 4 nodes using the other pattern). For this experiment we used a large data set\textsuperscript{7}. Since one of the patterns was not covering, the traversal of this non covering pattern had to be followed by the covering pattern so that all nodes could be moved into the new data block.

The parallel version of refresh.potential was tested on an x86 machine running Linux. The experiment utilized pthreads to handle the parallel work in a fork-join like way (emulated using mutexes to improve performance). Although a single branching point is not optimal from a parallelization point of view, it serves here to show that even a simple mechanism like this can have a substantial effect. The refresh potential test utilized the large reference data set from the MCF code.

### 4.6.1 Refresh Potential Optimizations

The results for the refresh potential function optimizations are shown in table 4.5. The table shows how often restructuring was done, the time for the refresh potential function, the overhead from restructuring and speedup of the function. Note that the program’s total execution time went up in all cases

\textsuperscript{6}Data set named \textit{train} distributed with mcf.

\textsuperscript{7}Data set named \textit{ref} distributed with mcf.
4.6. RESULTS

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Base (s)</th>
<th>Overhead (s)</th>
<th>Speedup (%)</th>
<th>Speedup Idx (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base</td>
<td>1.55</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>100</td>
<td>0.9</td>
<td>1.58</td>
<td>-37.5</td>
<td>46.5</td>
</tr>
<tr>
<td>200</td>
<td>1.1</td>
<td>0.8</td>
<td>-18.4</td>
<td>31.4</td>
</tr>
<tr>
<td>400</td>
<td>1.2</td>
<td>0.43</td>
<td>-4.9</td>
<td>24.7</td>
</tr>
<tr>
<td>800</td>
<td>1.25</td>
<td>0.21</td>
<td>6.2</td>
<td>22.0</td>
</tr>
<tr>
<td>1000</td>
<td>1.27</td>
<td>0.17</td>
<td>7.6</td>
<td>20.4</td>
</tr>
<tr>
<td>1100</td>
<td>1.31</td>
<td>0.14</td>
<td>6.9</td>
<td>17.1</td>
</tr>
</tbody>
</table>

Table 4.5: Optimizations targeting refresh potential

<table>
<thead>
<tr>
<th>Machine</th>
<th>Serial</th>
<th>Parallel</th>
<th>Speedup (tot ; func)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Core 2, 2.66GHz</td>
<td>130.3 s</td>
<td>122.4 s</td>
<td>6.1 % ; 13.6 %</td>
</tr>
</tbody>
</table>

Table 4.6: Parallelization of refresh potential. Total execution time.

(this is not shown in the table) as the restructuring caused other functions to run slower as they used different access patterns, the improvements in the tables are solely for the execution time of the refresh_potential function. The overhead column shows the overhead from the restructuring when hash tables were used to keep track of pointers. The indexing version was 10 times faster at restructuring than the hash table version (only the derived speedup is shown in the table).

4.6.2 Price Out Impl Optimizations

The graphs in figures 4.15, 4.16 and 4.17 show the speedup obtained on different platforms using the optimized MCF data structures. The different horizontal axes indicate the interleaving levels for the two different access patterns. The primary pattern is the covering access pattern and the secondary access pattern is the one responsible for optimization (i.e. the pattern used in price out impl). The two axes together mesh out the speedup on the vertical axis.

4.6.3 Parallelized Refresh Potential

The parallelization tests where run on a quad 2.66 GHz, Core 2 machine. Table 4.6 shows the average execution time during 20 runs on this machine. The speedup is reported for whole program and the refresh_potential function.
Figure 4.15: Core 2 @ 2.66 GHz, 4 GiB RAM, 6 MiB L2, Linux

Figure 4.16: i7 @ 2 GHz, 4 GiB RAM, 6 MiB L2, OS X Lion
For the initial experimentation with the refresh potential function we see that when using index based vectors the optimization quickly pays off (locally). Although the total execution time was higher for the program, the approach is still valid as many applications will not show the same behavior. In fact, the experiment confirms that substantial speedups of certain types of applications are feasible with the approach laid out in this chapter.

For the second experiment with the price out impl access pattern, the optimization experiment demonstrated that the approach is feasible, has a total performance improvement on the application and is applicable to more complex programs. This improvement was seen on all the platforms we tested. Although this improvement was very small on the Core i5 which we guess is due to the limited amount of cache in the processor. However, this should have been expected as the reordering is primarily a cache optimization of the data. For the experiment in particular, only the secondary access pattern should be used, though the primary pattern is still necessary to make a complete restructuring.

We also showed how a minor rewrite of the refresh potential function could
enable semi-automatic parallelization of the function. This is substantial in itself from a conceptual level, and even as the performance was improved (by 6.1 % for the whole program and 13.6 % for the function by itself). The method used only one extra worker thread and additional parallelism remains unexposed. While there are programming models that allow the programmer to express recursive parallelism such as the mentioned cilk, for many similar codes the runtime overhead would simply be too large. The Pax C extensions enable similar optimization in an entire program by concentrating the modifications on a few data types instead of in the code. Better hardware support such as proposed by the microgrid team [6], could possibly be applied to relieve the situation for these codes.

4.8 Conclusion

In this chapter we presented extensions to the C programming language that enable the restructuring of large pointer-linked data structures. The restructuring step converts all pointers into either fat pointers, slices or indices. This rewrite of data structures enables position independence for pointer-linked data structures, and the reordering of objects into a more optimal memory layout. It is feasible to apply compile-time checking of the validity of the pointer attributes in many cases, and, when not, it is possible to resort to runtime checks.

The restructuring done with the help of the language was demonstrated to give noticeable performance improvements for some common problems involving sparse matrices and spanning tree algorithms, and this without the utilization of the additional aliasing information that the attributes make available to the compiler.

In general it would be nice to express any kind of property using the most generic language possible. However, while it is of interest to have a general programming model for describing data structures, the fact is that there is a significant tradeoff between expressiveness and programmability. A language extension must be usable for a human programmer. There is thus a tradeoff between a purely theoretical model and a more practical approach that is easy to learn and work with. For this purpose, we have tried to avoid defining a too generic system that is difficult to use, and a too verbose and extensive system that is difficult to learn. We aimed at a set of attributes and language extensions that is minimal but still functional, and although the balance between ease of use and generality is very much subjective and is difficult to quantify in a testable way, we believe that by keeping the set of attributes minimal
but still functional the Pax C extensions described in this chapter manages to strike a good balance between ease of use and generality.