Interprocedural Analysis of Aggregates

Martin Jambor
Charles University
jamborm@matfyz.cz

Abstract

Currently, the main method of making contents of aggregate members available to other optimizations is by scalar replacement which requires the aggregate is allocated on stack and is its address is not shared among different functions. The latter condition can sometimes be fulfilled by inlining but the number of aggregates which need to live in memory is big even when inlining aggressively. This paper discusses design and implementation of an interprocedural analysis of aggregates which is capable of identifying aggregates which are used in a simple way so that they do not have any aliases. Moreover, the exact way how references to such aggregates are passed in between functions is also determined. We have also proposed interprocedural propagation of constants within aggregates on top of this analysis to demonstrate its usage. In the experiments we have carried out, the analysis was able to detect that about 20% of aggregates and pointers and references to them were used simply enough and thus can be reasoned about easily.

1. Introduction

An increasing number of members of the scientific community embrace C++ and object oriented programming (OOP) in general because it increases maintainability, reusability [11] and interoperability and is better suited to rapid application development than the traditional Fortran. C++ has also other advantages such as its much bigger user base which for example means that features of operating systems and libraries are often available to C++ programmers much earlier before Fortran users can enjoy them [17]. Naturally, the performance of the compiled code is still of primary importance. In particular, techniques like expression templates [22] and template metaprograms [23] together with better C++ compilers enabled users C++ programs to match or even exceed the performance of highly optimizing Fortran 77 ones [24, 17, 21, 6]. Nevertheless, the efficiency of the code produced is obviously all the more so dependant on the quality of the compiler [4] and there is still room for improvement.

Probably the best known reason why Fortran compilers can safely perform some optimizations which are difficult to prove legal in C++ are aliasing rules of procedure arguments. While the Fortran language standard dictates that global variables and subroutine arguments never alias, C++ must rely on sophisticated alias analyses to prove such properties. Another well-known difference between the two languages are much tougher operator reordering rules of C++ which must take into account such issues as overflow and underflow exceptions [7]. Finally, Fortran mathematical arrays are elementary types whereas they and particularly the operations on them must be provided by a library in C++ to provide the same or similar semantics [6]. There is a number of reasons why the operations implemented by such a library may not be as efficient as those on Fortran arrays. One of them is that most optimizations are designed to work only with simple types and these arrays are often represented by objects and thus opaque aggregates. Another is that the compiler may not for example be aware that some important properties such as sizes and strides stored within these objects representing arrays are available at compile time due to the many abstraction levels involved. Moreover, these problems are not specific to implementation of arrays or mathematical structures in general. Any analysis the compiler performs to comprehend these arrays representations is likely to be beneficial when used on implementation of more complex entities in scientific computing and beyond.

In this paper, we propose an *interprocedural analysis of aggregates* that is able to identify structures and arrays which, even when shared among a number of functions, never leave the current compilation unit and cannot be uncontrollably aliased and thus which are under full control of the compiler. Moreover, it captures which aggregate variables in different functions actually refer to same objects. In order to demonstrate how such analysis can be used, we have implemented interprocedural constant propagation of values stored in scalar and array aggregate members on top of it [14].

We start with Section 2 in which we will briefly describe how aggregates are dealt with today and specify what aggregates we are after. Section 3 and 4 contain implementation overview of the interprocedural analysis of aggregates and the interprocedural propagation of constants within aggregates that is implemented on top of it. Section 5 discusses experimental results and section 6 concludes.

2. Motivation

2.1 Status quo of optimizing aggregate members

The traditional way of removing layers of abstraction is procedure integration, also known as inline substitution [18]. Many C++ programs and especially those relying on expression templates contain huge numbers of tiny procedures which incur great call overhead and offer very little room for any optimization on their own. Replacing a call to such functions with a copy of its body therefore not only removes the overhead but also enables other intraprocedural optimizations that need bigger scope and has other beneficial effects on such things as scheduling and register allocation [3].

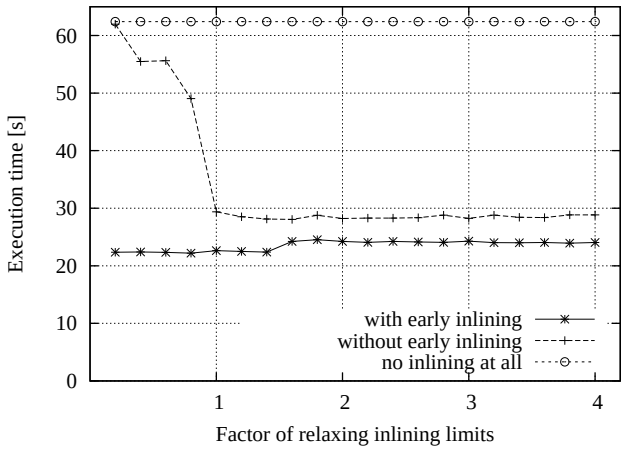

Figure 1. Effect of inlining on execution time of Tramp 3D.

GCC features both traditional inlining and so called early inlining which differ in how they select call sites for substitution, both are thoroughly described in [12]. In order to evaluate their efficiency we ran a series of experiments [14] in which we gradually relaxed inlining limits of GCC trunk development version[1] and measured the effect on execution and compile times of Tramp 3D benchmark [10]. The effects on execution time are presented in figure 1. The x-axis describes how many times the inlining limits determining the maximum growth of functions and compilation units were relaxed. 1 represents the default compiler setting, 3 means both limits were tripled and 0.2 means they were divided by five. The Tramp 3D benchmark extensively exploits expression templates which produce great numbers of tiny functions and thus the effect of inlining on the execution of the benchmark is big. As you can see, the non-inlined executable is almost three times slower than the versions which we compiled with relaxation factor greater than one. On the other hand, another important observation is that from some point on, further relaxation does not buy any performance increase and in fact can degrade it a little. Having said that, it is important to state that relaxation of inlining limits led to linear increases in compilation time [14]. For example, when inlining limits were four times more permissive than the default, the compilation took twice as long with early inlining and three times as long without it. In summary, inlining is a powerful technique but it has its limits and making it excessively permissive brings no benefits but has substantial drawbacks.

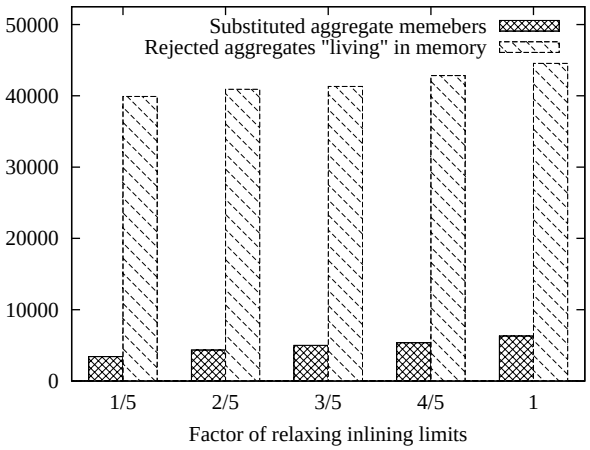

Figure 2. Effect of inlining on the number of instantiated scalar replacements of aggregate members and aggregates rejected by SRA when compiling Tramp 3D benchmark. Please note that SRA pass was run twice so many of the rejected candidates were reported twice.

If all functions using a stack-allocated aggregate are inlined, it may become a candidate for *scalar replacement of aggregates* (SRA) [18]. This transformation breaks up structures and arrays into individual scalar components because most optimization passes cannot work on aggregates. However, aggregates must comply with several conditions in order to be considered by SRA. Most importantly, the aggregates must not be aliased with anything else which is guaranteed by requiring they do not need to live in memory[2]. This requirement is very strong, it for example means they have to be local variables and cannot be passed by reference to other functions. As more functions are inlined, the number of successful replacements slightly increases but the number of candidates deemed unsuitable because they need to live in memory increases as well (see figure 2). Finally, SRA does not even look at aggregates accessible through pointers such as `this` in C++.

2.2 Goals of the new analysis

Therefore we have designed and implemented *interprocedural analysis of aggregates* which is capable of identifying aggregates that are shared between different functions but which are used simply enough to rule out any aliasing and track how such aggregates are passed in between different functions. Consider the following simplified example.

[1] Revision 123774.

[2] i.e. `is_gimple_non_addressable()` returns `false`.

```
void example()
{
  Array2D A(600, 800);
  Array2D *B = new(600, 800);

  A.acquireContents();
  B->loadFromFile(f);
  B->addElementWise(A);

  A.storeOutput();
  B->storeOutput();

  delete B;
}
```

Fields of the array objects inevitably contain such potentially useful information as sizes and strides. Currently, unless all methods of `A` that are (even indirectly) invoked from `example` function are inlined, no optimization passes can access these information about array `A`. Moreover, since `B` is dynamically allocated, it will never be considered by SRA. On the other hand, methods of both objects access them only through `this` pointer which is never stored elsewhere and so there is no danger of aliasing. The primary goal of the interprocedural analysis of aggregates is therefore to identify well-behaving objects (structures and arrays) which comply with all following conditions in all functions they are passed to:

1. The aggregate is either passed to the function as a parameter or is allocated on the stack or the heap (through malloc-like functions or the built-in `new` operator).
2. The address of the aggregate or any of its parts does not escape to a global variable, field in a structure or an element of an array and does not take part in any inline assembler statement.
3. No pointer or reference to the aggregate or its part is passed to a function which could not be analysed. The only exceptions are deallocation functions such as free and built-in `delete` operator.
4. If a pointer or a reference to an aggregate or its part is passed to a known function, no other reference to the aggregate or any of its parts is passed to the same function through a different actual parameter.
5. If an aggregate is accessible through a pointer or a reference, such pointer or reference variable is not engaged in any phi node. This means it is neither its parameter nor its result.

The first four conditions guarantee there are no aliases to any of identified well-behaving objects. The fifth one ensures we know what variable refers to what aggregate at all times. It is evident that `this` pointers comply with these requirements if they are used only to access object's fields and call its methods defined in this compilation unit.

The analysis also creates and keeps a mapping from local variables, actual and formal parameters to structures describing aggregates. The pass needs it in order to perform its job but it is also essential for other transformations using this analysis to find out what entities they have on their hands and how they are passed in between different functions. For example, another interprocedural pass can use this information to find out what has happened to a particular field accessible through the `this` pointer across a call expression. For instance the propagation of modification flag described below can tell that it remains unchanged.

3. Implementation

The analysis is divided into the following stages:

1. *Preparation stage.* During this stage, various important data structures are created and initialized.
2. *Intraprocedural usability and constantness analysis.* The compiler examines all functions, one at a time, and assesses the usability of aggregates and pointers to aggregates and determines which members of these aggregates are modified[3].
3. *Interprocedural usability propagation.* The analysis then propagates the unusable flag along call graph edges to all (and even indirect) callers and callees.
4. *Interprocedural propagation of modification flags.* Similarly, the information that members have been modified is propagated to callers.

3.1 Data structures describing aggregates

The analysis examines structures, arrays and standalone local pointers and references to structures and arrays. It represents these aggregates, their members and pointers by a single structure called `ocp_item` (see figure 3). From now on, an *item* always means an instance of this structure. When an aggregate is passed between more functions, each function represents it with an item of its own and these items are connected by structures associated with call graph edges. Among other things, these items contain the following information:

- Pointers to the function and the variable or SSA name this item belongs to. There may be other pointers (SSA scalars) referring to the same aggregate or its part. Items corresponding to such aliases have their `target` pointer set to the associated (sub-)item.
- Children. Items representing structures keep pointers to array of items representing individual fields.
- Top-level objects. The `object` field points to the top-level item this sub-item is a child of. This can be a pointer to itself. Please note that this information is also local to

[3] The detection of modifications is not needed to satisfy conditions set in section 2.2 but proved to be useful and it is convenient to do it at this stage.

```
struct complex {
        double real;
        double img;
};

struct example {
        char *name;
        struct complex value;
        struct complex *other;
        int data[10];
};

struct example b, *a = &b;
struct complex *c = &b.value;
```

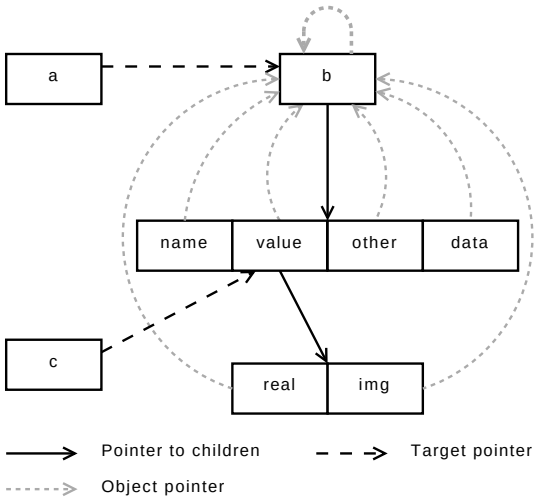

Figure 3. An example aggregate variables and their representation as items. The figure also demonstrates the meaning of `children`, `object` and `target` fields of an item.

the given function. If a reference to a sub-item is passed to another function, the callee considers the reference a top-level item.

- Two "next" pointers realising a forward and a backward queue (see section 3.3).
- Flags. We will discuss the most important item flags where appropriate throughout the section.

As you can see, the analysis is built on top of the new SSA framework for interprocedural optimizations [13].

3.2 Intraprocedural analysis

The second stage itself can be divided into three distinct parts. First, we examine the definitions of SSA names of pointers and references to interesting types. Second, uses of the very same names are scrutinized. Finally, the analysis scans the whole function for calls of other functions and operations on local aggregates (i.e. not pointers). If we stumble across any operation that is not deemed safe, the corresponding item is flagged as *unusable*.

As far as SSA name definitions are concerned, definitions by phi nodes are immediately discarded as unusable to satisfy the condition 5 in section 2.2 and default definitions are considered fine without any further checks. Definitions by assignment statements require further attention depending on the type of the right hand side:

- `ADDR_EXPR`. The function analyzes its only operand and if it corresponds to another item or any of its sub-items, it sets the target of the defined SSA name appropriately. If it does not, the item is marked as unusable.
- `CALL_EXPR`. The item defined as a result of a function call is usable only if the function is some kind of malloc (as explained below).
- `SSA_NAME`. If one SSA name is defined by means of another, there are two possibilities. If the name on the right hand side corresponds to an item, it or its target becomes the target of the newly defined name. Furthermore, the pointer on the right hand side can be a void pointer holding a result of a malloc-like function having no other uses except for this statement. Any other case means the item in question is marked unusable.
- `NOP_EXPR`. Type conversion is the most complex case which handles similar cases to the three above and attempts to locate a sub-item of the requested type at the beginning of the located targets.
- If the right side does not fall into any of the categories above, the defined SSA name is marked as unusable.

Once definitions of SSA names have been dealt with, it is necessary to make sure their uses do not leak addresses. Therefore, all uses of all SSA names that have not been rejected in the previous step are examined one by one. If a phi node is encountered, the condition 5 in section 2.2 requires the corresponding item is marked as unusable. We handle functions and `ADDR_EXPR`s later on when traversing the whole function, and so it is only necessary to examine assignment uses at this stage. The left hand side is inspected to determine whether this statement is a definition of a known local alias of the given item. If it is so, the statement is fine and the check of this use can be terminated. Otherwise, the right hand side is decomposed and searched for any unedereferenced uses of the SSA name that is being checked. If one is found, the address of the corresponding object escapes our control and thus the associated item is marked unusable. If the use is indeed valid, we also examine the left hand side to find out whether it modifies any of the members of the

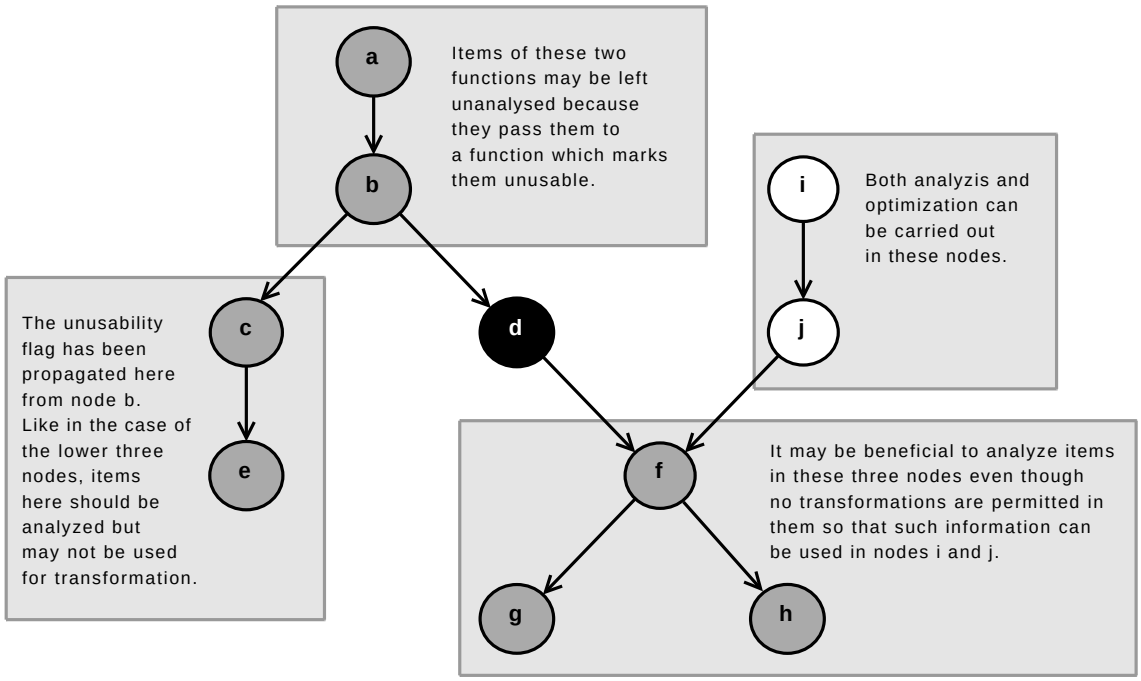

Figure 4. Propagation of the unusability flag in a call graph. Assume each function has one parameter and passes the object it gets from callers to all of its callees and function d marks the associated item unusable.

given item as a part of the constantness analysis. If it does, the sub-item is flagged as modified.

Unfortunately, the dataflow information available for SSA names of local pointer variables is not there to help us reason about what happens with local aggregates. On the other hand, aggregates themselves cannot escape, only their addresses can. We therefore need to locate all `ADDR_EXPR`s obtaining addresses of these aggregates or any of their parts and mark its item as unusable if the address is not stored into a trusted SSA name. That is why the last step in this stage is scanning the whole function and examining each statement whether it somehow contains such address expression. While we are at it, it is also reasonable to process calls to other functions, because we need to determine which items are passed as actual arguments to them in order to create a mapping between formal and actual parameters along call graph edges so that usability and other information can be propagated in between corresponding caller and callee items. Finally, all items passed to a function that cannot also be analysed are marked as unusable because their address can escape our control.

The objects representing scientific entities are rarely ever passed to standard library functions and their addresses are almost never returned by any functions at all. The important exceptions are functions for dynamic allocation and deallocation of objects, in C++, the operators `new` and `delete`, in C, standard functions `malloc()` and `free()`. In order for the analysis to consider heap allocated aggregates, we must detect and allow these singularities. Malloc-like functions are flagged with `ECF_MALLOC`. Because there is no such flag for free-like functions we resort to identifying built-in `free()` instead. Recognizing `new` and `delete` is another matter because they are not built-in functions but rather a part of libstdc++ where they are defined by a couple of C++ source code lines. At the moment we detect them using two new special purpose attributes. Naturally, a more general approach will probably be required in the future.

3.3 Interprocedural usability propagation

When an item happens to be marked as unusable in one function, for example because the address of the associated object is stored in a global variable or another uncontrollable place, this information obviously needs to be propagated to other functions which work with the same object so that no transformations requiring total control over the item takes place there. Objects are passed by reference and therefore no matter where aliasing may potentially occur, all direct and indirect callers as well as callees must refrain from carrying out such optimizations. On the other hand, sometimes it may be useful to carry on with analysis in the callees nevertheless.

Consider the example of a call graph shown in figure 4 [14]. Assume each function has one parameter and passes the object it gets from callers to all of its callees and function d marks the associated item unusable. All functions which may at some point work with the object that was made unusable by d must not perform any optimization on it. Obviously, the functions f, g and h fall into this category because they receive their parameter from d. It must also

include the functions a and b because they may continue to work with an object that has already been passed to d. Perhaps a bit more intriguingly, we must prevent optimizations in functions c and e too because function b might have given them an object after it had passed it to d and which is thus unreliable. On the other hand, nodes i and j can consider their items safe, because there is no way they are passed to d even though they call a non-optimizing function. That also means there are functions like f, g, h, c and e which may not perform optimizations based on items themselves but should gather information during the analysis stage nevertheless because it might be useful to other callers (functions i and j in the example).

In order to propagate appropriate information to all necessary places, the interprocedural stage performs a so called *forward unusability propagation* and *backward unusability propagation*. Both propagations simply pick up items from a corresponding work queue and add items of any callees or callers (respectively) they find necessary to *both* queues. Specifically, forward propagation only propagates unusability of items from callers to all callees that are passed a reference to it. On the other hand, backward propagation not only spreads the flag to all callers but to callees as well (so that optimizations are prohibited in functions like c and e in the figure 4). Both propagations alternate until both queues are empty. Item flags store information whether the unusability was propagated to the item by forward or backward propagation or both. This is because items which have not been marked unusable by the backward propagation should still be analysed by transformations using this analysis because there may be other callers of these functions where corresponding items are perfectly suitable for optimizations (e.g. functions f, g and h in figure 4). As one would expect, we initialize both queues with items found unusable in the intraprocedural stage. These items have all three unusability flags set because they cannot be subject to either transformations or analysis.

Finally, as a transitional step in between this analysis and those that build on it, we propagate the modification flag of all sub-items from callees to callers by a very similar mechanism.

4. Interprocedural Propagation of Constants within Aggregates

We have written an interprocedural analysis of aggregates not only to serve as an example how the interprocedural analysis of aggregates might be used but also as an attempt to make useful information stored in objects such as sizes and strides discussed in section 1 available to subsequent optimizations if they are available at compile-time. The propagation closely follows the traditional approach [5] and uses the well-known lattices [25, 18] that were extended to be able to store up to sixteen constant values stored at known constant indices in an array [14]. Another notable difference stemming from the fact that aggregates are passed by reference is that we need to be concerned not only with *jump functions* but also with *return functions*. The propagation thus operates in four stages:

1. *Jump and return function building stage.* This intraprocedural phase calculates jump functions describing actual arguments of each call site within a function and builds the return function of the each procedure.
2. *Interprocedural lattice analysis.* The information obtained in the previous stage is put to interprocedural use. A lattice item is assigned to each usable member of each aggregate formal parameter of each function.
3. *Replacement stage.* The compiler replaces uses of any member which was discovered to be constant with the appropriate constant.
4. *Cleanup.* Finally, the pass frees all memory that is no longer used.

4.1 Jump and Return Function Building

We have decided to use the *pass through* functions as described in [5], recommended by [9] as the most cost-effective, and used by the scalar interprocedural constant propagation pass that is nowadays part of GCC [15]. Such function can tell that a particular leaf item:

- has an unknown, potentially variable value (represented by bottom),
- has a known constant value (this case is represented by the constant itself) or
- has the same value as it had when the caller was invoked. (This case is very conveniently represented by top.)

We have already stressed that because the objects we analyze are passed by reference, we also must construct return functions that describe what happens to an item when it is passed to a particular function. This function denotes that a particular leaf item:

- was potentially redefined with an unknown or variable value (this is also represented by a bottom),
- was in all cases redefined with a known constant (represented by the constant) or
- was left intact by all possible control paths of the given function (represented by top).

The scalar interprocedural constant propagation that is currently part of gcc performs intraprocedural data flow analysis on the SSA form [12, 13, 15, 2] and thus avoids the need to scan the whole function. However, such information is not available for aggregate members and so we had to resort to the good old iterative approach [25, 18]. We therefore keep lattices describing values of all scalar sub-items of all usable aggregates at the beginning of each basic block. Initially, elements corresponding to the entry basic block and

formal parameters are initialized to top, those representing the aggregates and dynamically allocated entities at the entry block are set to bottom. The analysis is driven by a yet another work list initialised to contain the entry block. The analysis then picks up a block from the list, traverses it to reflect the effect it has on the examined aggregate members and propagates this information to lattices corresponding to all succeeding basic blocks. If a lattice is modified, the associated basic block is added to the work list. The analysis finishes when the work-list is empty. At this point, the lattices at the exit block contain the return function of the current call graph node. The analysis takes extra care to include the potential effects of exceptions by acting as if there were edges from instructions that may throw an unhandled exception to the exit basic block.

The pass constructs jump functions whenever a call expression is encountered while traversing a basic block. Moreover, we also combine the current value of the lattices with the return function of the callee, if available. In order to have as many return functions of callees at our disposal, we perform this interprocedural analysis in topological order. Still, cycles in the call graph cause situations where return functions have not been computed. In these cases we check the modification flag of all relevant sub-items and if is set, we set the lattice corresponding to the particular sub-item to bottom (see example in figure 5).

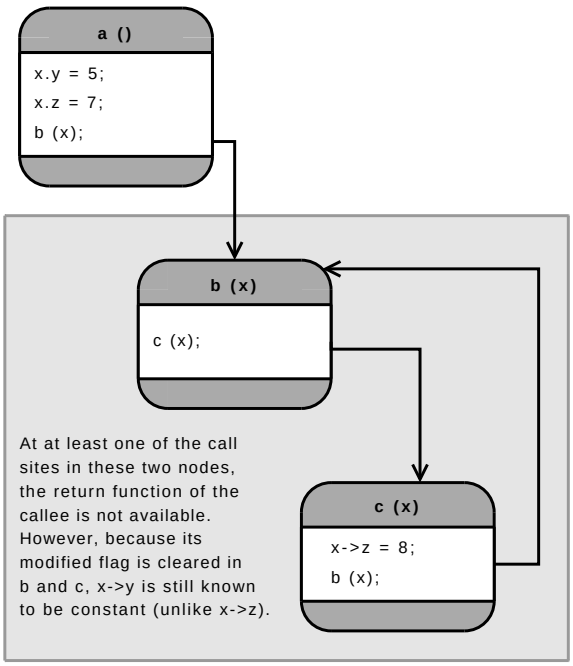

Figure 5. Return functions, `modified` flag and recursion.

4.2 The rest of the propagation

Having the jump functions at our disposal, we must use them to propagate the information all over the call graph. This mechanism is well described in Callahans's paper [5]. The algorithm keeps a work list of call graph nodes which is initialised by pushing all call graph nodes on top of it. When an item is popped from the list, the current values of lattices of interesting sub-items are propagated along the jump functions and call graph edges to all callees that obtain these items intact. If a lattice of an item in another node is altered, the node is added to the work list again so that its callees are potentially updated and so on until the work list is empty.

If some items corresponding to a part of a formal parameter are discovered constant on all invocations, the propagation creates a new clone of the function and replaces all untampered uses of the given aggregate member by the revealed constant.

It remains to be shown how we make sure the resultant code is correct even in presence of calls from outside of the current compilation unit. All callers of the old function in the current compilation unit are redirected to call the new copy, which is then altered. Therefore, if the function is called from outside of this module, the old unaltered copy will be invoked which itself will definitely cause no problems. On the other hand a function called from outside of this module can cause trouble indirectly by calling another, altered one, in an unexpected context. Consider the situation in figure 6. Node b can be called either from a with x equal to a constant or from outside of this module with any possible value. b passes x to c and the interprocedural lattice analysis propagates the constant there too. When uses of x are substituted in a clone of b, that clone is inaccessible from outside of the module and thus safe. However, the original node b obviously must not call the modified version of c which assumes x is always equal to four but the original version of c too.

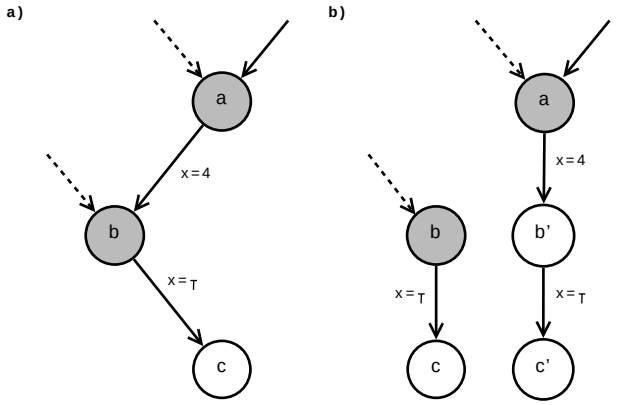

Figure 6. Call graph modifications. The grey nodes are visible outside of this compilation unit. **a)** The situation before substitution, **b)** after cloning and call graph modification.

Therefore, the last step in this stage is *call graph update*. During this phase we find all call graph edges from an original node to a cloned one that have a top lattice in any of the jump functions and redirect them to the original node. In the end, there is no path from an original node to a cloned

one on which an item would be passed along the whole way because it is passed only when all jump functions are tops. Consequently, no modified function is ever called with a parameter received from outside of this module.

5. Results

5.1 Benchmarks and statistics

Table 1 shows how many unique aggregates that comply with conditions set out in section 2.2, usable items (i.e. aliases of the complying aggregates) and unusable items the interprocedural analysis of aggregates identified while compiling Tramp3D [10], FreePOOMA test suite [1] and Xpdf [20]. Let us reiterate that all local objects, structures and arrays and local pointers to such types do have an item associated with them. This means that the analysis is able to provide information about more than quarter of such local variables in Tramp3D.

Benchmark	Aggregates	Usable Items	Unusable Items
Tramp3D [10]	2809	9174 (27%)	26143 (73%)
FreePOOMA test suite [1]	67311	162845 (23%)	558497 (77%)
Xpdf [20]	1134	1719 (15%)	9495 (85%)

Table 1. Number of individual aggregates and usable and unusable items.

Table 2 shows the number of replacements the interprocedural constant propagation within aggregates performed when compiling various benchmarks. All compilations listed were done using the default configuration, i.e. with the default set of switches and parameters for the particular project. Above all, that means interprocedural scalar constant propagation was not switched on which might have hindered some opportunities to propagate constants within aggregates.

Benchmark	Scalar	Array	Total
GCC bootstrap	189	0	189
GCC test suite	675	12	687
FreePOOMA [1] test suite	3367	12	3379
Tramp3D [10]	2	0	2
DLV [16]	118	0	118
Blitz++ [21] test suite	96	0	96

Table 2. Number of replacements performed.

In order to measure the impact of the transformation on execution time of C++ scientific applications, we have run a number of benchmarks including Tramp3D [10], DLV [16], and Blitz++ [21] acoustic 3D benchmark. In all these cases, we have configured GCC to optimization level 2 with loop unrolling, scalar interprocedural constant propagation and static linking. Some of the results are presented in table 3.

All benchmarks presented in this section have been carried out on an AMD Athlon 64 Processor 2800+ running in 64bit mode and equipped with 1GB of RAM. We took all time measurements three times and present here their arithmetic mean, although all corresponding measurements gave very close values. The pass was running within GCC 4.3, which is currently under development, specifically we used the subversion revision 123774.

Benchmark	Unpatched	Patched
Tramp 3D	11m7.27s	11m6.55s
Blitz++ acoustic 3D	7.25s	7.14s
DLV primeimpl2	5.89s	5.78s
DLV ancestor	86.82s	82.74s
DLV 3col-simplex1	4.26s	4.17s
DLV 3col-ladder	97.37s	93.72s
DLV 3col-random1	6.89s	6.77s
DLV hp-random1	8.15s	8.08s
DLV decomp2	7.07s	6.87s
DLV bw-p4-esra-a	39.21s	38.61s
DLV bw-p5-nopush	3.64s	3.38s
DLV bw-p5-pushbin	3.24s	3.10s
DLV 3sat-1-constraint	9.99s	9.77s
DLV ramsey	3.93s	3.62s
DLV cristal	6.38s	6.09s
DLV hanoi-k	17.23s	16.62s
DLV mstdir	7.12s	6.99s
DLV mstundir	70.97s	69.99s
DLV timetabling	5.00s	4.72s

Table 3. Execution times of various benchmarks.

We did not encounter any issues with excessive run time or memory consumption when running these experiments. For example, the pass takes less than 1% of the total runtime of the compiler and allocates slightly over 1MB when compiling Tramp3D.

5.2 Optimizing template expressions

We have already stressed that C++ scientific calculations rely on template expressions [22] and thus we spent some time investigating how interprocedural aggregate analysis and the constant propagation built on top of it treat them. Unfortunately, expression templates perform a number of operations that the analysis cannot cope with. These expressions rely on templates to produce complex objects representing a given expression, its type, operands and internal tree structure, which are then capable of computing final and intermediate results. These objects

1. contain references to operands in their internal fields,
2. the addresses are copied down the tree of these objects and
3. once the objects are full constructed they are regularly passed back using the return statement.

Unfortunately, the analysis will not be able to do much with code resulting from expression templates until it is able to accommodate at least simple forms of these three cases. We also believe that the effect our transformation has on execution time is rather modest exactly because many of

the important objects such as arrays were deemed unusable because they take part in expression templates. Nevertheless, providing for code constructs described above remains to be future work.

6. Conclusion

In this paper, we have described the current approach to making aggregate members available for optimization and shown its effects and limitations. In order to allow further optimizations, we have proposed and implemented so called interprocedural analysis of aggregates to overcome this problem for aggregates which can be easily proved to be fully under compiler's control. The paper describes exactly what conditions such aggregates and references to them must adhere to and overviews the data structures that are used to represent such variables and their relationships. The implementation principles have also been outlined. In order to demonstrate the use of the new analysis, we have also developed modification flag propagation and an interprocedural propagation of constants within aggregates. Again, we have explained the basics of implementation of both. Finally, we have discussed results of both the analysis and the constant propagation on a number of benchmarks and identified features required to improve its handling of expression templates.

Apart from trying to adapt the analysis to cope with code such as that results from expression template, we are also currently exploring potential uses. In addition to propagating constants it can be also used for example to drive procedure cloning [8]. The propagated values may not only be compile time scalar constants but also for instance type information which is used when devirtualizing monomorphic [19] calls. The analysis can also help to determine which fields of a structure are never used during its lifespan and so can be removed from it. Furthermore, obtained data can be used for a special kind of alias analysis because a usable item cannot have any aliases. We are sure there are many other possibilities we have not thought off yet. Nevertheless, we would like to take into account as many of them as possible when adding the analysis into the development version of GCC.

Acknowledgments

I would especially like to thank Jan Hubička for his insightful and patient supervision. I would also like to thank Andrew Pinski, Steven Bosscher, Daniel Berlin, Diego Novillo, Richard Günther and others who have provided me with very relevant comments and enormously helped me to comprehend GCC internals.

References

[1] FreePOOMA. http://www.nongnu.org/freepooma/.

[2] GNU Compiler Collection source code. http://gcc.gnu.org/.

[3] Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 134–145, New York, NY, USA, 1997. ACM Press.

[4] Federico Bassetti, Kei Davis, and Daniel J. Quinlan. A comparison of performance-enhancing strategies for parallel numerical object-oriented frameworks. In *ISCOPE*, pages 17–24, 1997.

[5] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *SIGPLAN '86: Proceedings of the 1986 SIGPLAN symposium on Compiler construction*, pages 152–161, New York, NY, USA, 1986. ACM Press.

[6] John R. Cary, Svetlana G. Shasharina, Julian C. Cummings, John V. W. Reynders, and Paul J. Hinker. Comparison of C++ and Fortran 90 for object-oriented scientific programming. *Computer Physics Communications*, November 1996.

[7] C++ Standards Committee. Standard for programming language C++ - working draft.

[8] Keith D. Cooper, Mary W. Hall, and Ken Kennedy. A methodology for procedure cloning. *Computer Languages*, 19(2):105–117, 1993.

[9] Dan Grove and Linda Torczon. Interprocedural constant propagation: a study of jump function implementation. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 90–99, New York, NY, USA, 1993. ACM Press.

[10] Richard Günther. *Three-dimensional Parallel Hydrodynamics and Astrophysical Applications*. PhD thesis, Eberhard-Karls-Universität, 2005.

[11] Tom Houlder. C++ for scientific applications of iterative nature. *SIGPLAN Not.*, 32(3):21–26, 1997.

[12] Jan Hubička. Interprocedural optimizations on function local SSA form in GCC. In *Proceedings of the GCC Developers' Summit*, pages 75–83, Ottawa, Ontario Canada, 2006.

[13] Jan Hubička. Interprocedural optimization framework in GCC. In *Proceedings of the GCC Developers' Summit*, pages 57–68, Ottawa, Ontario Canada, 2007.

[14] Martin Jambor. Optimizations in the GNU Compiler Collection targeted at scientific computing. Master's thesis, Charles University in Prague, Faculty of Mathematics and Physics, 2007.

[15] Razya Ladelsky and Mircea Namolaru. Interprocedural constant propagation and method versioning in GCC. In *Proceedings of the GCC Developers' Summit*, pages 133–143, Ottawa, Ontario Canada, 2005.

[16] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Logic*, 7(3):499–562, 2006.

[17] Los Alamos National Laboratory. *Parallel Object-Oriented Methods and Applications (POOMA) Tutorial*.

[18] Steven S. Muchnick. *Advanced compiler design and*

implementation. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[19] Mircea Namolaru. Devirtualization in GCC. In *Proceedings of the GCC Developers' Summit*, pages 125–133, Ottawa, Ontario Canada, 2006.

[20] Derek Noonburg. Xpdf – A viewer for Portable Documentat Format (PDF) files. http://www.foolabs.com/xpdf/.

[21] Todd L. Veldhuizen. Blitz++: The library that thinks it is a compiler.

[22] Todd L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995.

[23] Todd L. Veldhuizen. Template metaprograms. *C++ Report*, 7(4):36–43, May 1995.

[24] Todd L. Veldhuizen and M. E. Jernigan. Will C++ be faster than Fortran? In *Proceedings of the 1st International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'97)*, Berlin, Heidelberg, New York, Tokyo, 1997. Springer-Verlag.

[25] Mark N. Wegman and Frank Kenneth Zadeck. Constant propagation with conditional branches. In *POPL '85: Proceedings of the 12th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 291–299, New York, NY, USA, 1985. ACM Press.