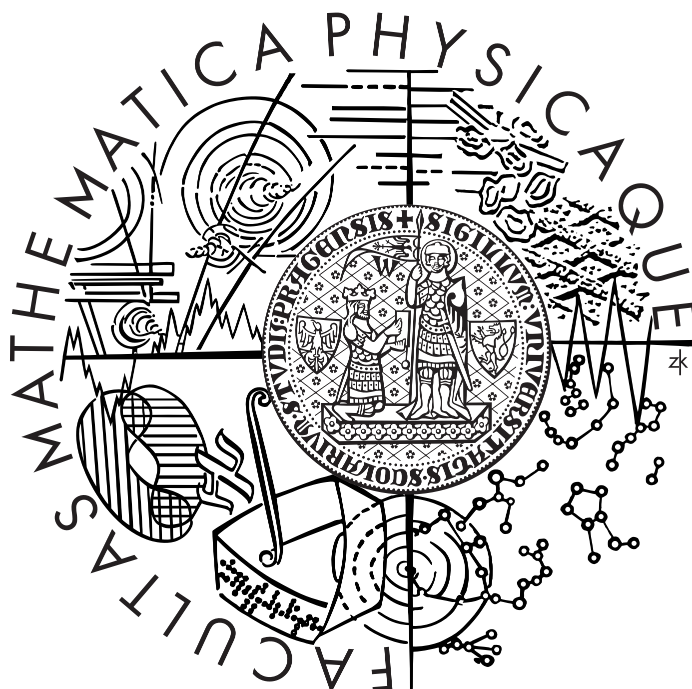
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

# Diplomová práce




**Martin Jambor**

# Optimalizace vědeckých výpočtů pro GNU Compiler Collection

Optimizations in the GNU Compiler Collection Targeted at Scientific Computing

Katedra Aplikované Matematiky
Studijní program: Informatika

Vedoucí práce: **Mgr. Jan Hubička**

Prohlašuji, že jsem svou diplomovou práci napsal samostatně a výhradně s použitím citovaných pramenů. Souhlasím se zapůjčováním práce a jejím zveřejňováním.

V Praze dne Martin Jambor

**Acknowledgments**

I would like to thank Jan Hubička, Andrew Pinski, Steven Bosscher, Daniel Berlin, Richard Günther and others who have provided me with very relevant comments and enormously helped me to comprehend GCC internals.

## Abstract

Title: Optimizations in the GNU Compiler Collection Targeted at Scientific Computing
Author: Martin Jambor
Department: Department of Applied Mathematics
Supervisor: Mgr. Jan Hubička

Abstract: Many members of the scientific community look for alternatives to Fortran to increase maintainability, reusability and interoperability of their projects and component and to achieve rapid development and deployment. C++ appears to be an ever more appealing alternative because evolving compilers and coding techniques continually boost the efficiency of the resultant code. This work describes what C++ scientific code typically looks like, and discuses a number of contemporary optimizing techniques compilers use to remove overhead caused by levels of abstraction. Moreover, it proposes a new Intraprocedural Analysis of Aggregates to expose even more information stored within objects and track object behaviour. It also describes implementation of intraprocedural propagation of constants within aggregates built on top of this analysis. Finally, it discusses its efficiency and potential for future work.

# Contents


**1 Introduction** **6**

**2 C++ Scientific Computing Techniques and Libraries** **8**

2.1 Expression Templates . . . 8

2.1.1 Implementing Array Expressions Without Templates . . . 8

2.1.2 Implementing Efficient Array Expressions with Templates . . . 10

2.1.3 Efficiency of the Resultant Code . . . 13

2.1.4 Other Uses of Expression Templates . . . 14

2.2 Language Extensions . . . 14

2.3 C++ Libraries for Scientific Computing . . . 15

2.3.1 Blitz++ . . . 15

2.3.2 Parallel Object Oriented Methods and Applications (POOMA) . . 15

**3 Existing Optimizing Techniques** **16**

3.1 Procedure integration . . . 16

3.2 Interprocedural Constant Propagation . . . 22

3.3 Procedure cloning . . . 23

3.4 Scalar Replacement of Aggregates . . . 24

3.5 Loop transformations . . . 26

**4 Interprocedural Analysis of Aggregates** **27**

4.1 Core data structures . . . 28

4.1.1 Data associated with aggregates . . . 28

4.1.2 Data associated with functions . . . 31

4.2 Preparation Stage . . . 33

4.3 Intraprocedural Usability Analysis . . . 33

4.3.1 Examining SSA names definitions . . . 33

4.3.2 Scrutinizing uses of SSA names . . . 34

4.3.3 Function scan . . . 35

4.3.4 Identifying malloc, free, new, and delete . . . 36

4.4 Interprocedural usability analysis . . . 36

4.5 Interprocedural Modification Flag Propagation . . . 38

4.6 Lattices and Their Internal Representation . . . 38

4.7 Jump and Return Function Building . . . 39

4.7.1 Dataflow from the global perspective . . . 41

4.7.2 Processing individual statements . . . 42
4.8 Interprocedural Lattice Propagation Stage . . . 43
4.9 Replacement Stage . . . 44
4.9.1 Cloning and externally visible functions . . . 45

**5 Results** **47**
5.1 Correctness . . . 47
5.2 Interprocedural Analysis of Aggregates and Expression Templates . . . 47
5.3 Benchmarks . . . 49

**6 Conclusion** **52**

**A Naive Implementation of Array Expressions in C++** **54**

**B Using Expression Templates to Implement Array Expressions** **59**

**C Demonstration of Template Expression Code Efficiency** **71**

**D Contents of the Supplemental CD** **72**

# Chapter 1

# Introduction

Until the early 1990s, the scientific community almost exclusively used Fortran 77. Nevertheless, several shortcomings of this language have made many to look for alternatives such as C++. The most pressing needs were ever more important requirements to improve code maintainability, reusability [22] and interoperability, allowing rapid application development, quick deployment and fast adaptation for specific problems and environments [21]. C++ has proved it tackles all these issues very well by providing well known object oriented programing techniques often described as encapsulation, inheritance and polymorphism [38]. Fortran community has tried to address the same issues with new standards of the language, namely with Fortran 90 and Fortran 95. These versions have managed to achieve encapsulation but struggle to provide inheritance and polymorphism [13]. Moreover, the Fortran's user base is shrinking, especially when compared to C++. New operating system interfaces and middleware are usually made available to C++ programmers much earlier before people can start using them from Fortran [29].

The two main arguments in favour of Fortran are the existence of legacy libraries and applications and its excellent performance [5]. On the other hand, newly invented algorithms require that old code is rewritten anyway and thus the importance of legacy software is continuously being reduced over time. Moreover, techniques such as expression templates [34] and template metaprograms [35] together with better C++ compilers enabled users of this language to match or even exceed the performance of highly optimizing Fortran 77 compilers [36, 29, 33, 13]. Still, the efficiency of the code produced is obviously all the more so dependant on the quality of the compiler [10] and there is still room for improvement. Nevertheless, a lot of effort is perpetually put into further enhancement of available compilers, particularly because C++ is such a popular language.

Probably the best known issue that C++ compilers must deal with, unlike Fortran ones, are relaxed aliasing rules. Fortran language specification disallows aliasing of arguments as well as aliasing between common (global) and dummy arguments in procedures [1]. C++ does not have this restriction and thus its compilers must either refrain from performing certain optimizations if they cannot guarantee the objects involved are not aliased by some, usually interprocedural, analysis. Another restriction C++ faces are much tougher operator reordering rules. Unlike Fortran's, its associativity and commutativity requirements take into account such issues as overflow and underflow exceptions [15]. For

example, $((a+10)-b)$ cannot be safely converted into $((a-b)+10)$ because if $(a+10)$ triggers an overflow exception but $((a-b)+10)$ does not, the two expressions are not semantically equivalent according to the standard. Moreover, overloaded operators are never considered associative or commutative. Additionally, Fortran provides standardized language support for parallelism and vector operations [31]. It also often emits much more meaningful error and warning messages [5].

Another key advantages Fortran compilers have over those crunching C++ code is that mathematical arrays are elementary types in Fortran whereas they and particularly the operations on them must be provided by a library in C++ [13]. The operations implemented by such a library may not be as efficient as those on Fortran arrays because the compiler may not for example be aware that some important properties such as sizes and strides are available at compile time due to the many abstraction levels involved. Conversely, arrays implemented by a library may be more flexible and suitable for a particular task. Nevertheless, in order to match Fortran's performance, the compiler must incorporate more sophisticated methods to analyse the properties of given objects and use them to produce efficient code. In addition, the libraries themselves must be carefully crafted so that even though they provide a lot of abstraction, its overhead is low and such that it can be easily removed by a compiler. This work explains what C++ scientific code typically looks like, describes some of the existing compiler optimizations aiming to reduce the abstraction overhead and finally proposes a new one specifically designed for object oriented scientific applications.

The rest of this work is structured in the following way. Chapter 2 explains basic source-level C++ techniques exploited by scientific libraries and applications and then briefly introduces a few selected C++ libraries for scientific computing. Chapter 3 deals with some existing compiler techniques used to optimize away the overhead imposed by abstraction and those essential to compile any scientific code well. Chapter 4 describes a new method of interprocedural analysis of objects and other aggregates passed in between different functions and continues discussing its implementation in the GNU Compiler Collection (GCC) together with a pass for interprocedural constant propagation within these aggregates. Chapter 5 presents results obtained with this technique and chapter 6 concludes.

| **Feature** | **C++** | **Fortran 90** |
|---|---|---|
| Encapsulation | available | available |
| Inheritance | available | achievable |
| Polymorphism | available | achievable |
| Templates | available | missing |
| Pointer casting | available | missing |
| Exception handling | available | missing |
| Built-in math arrays | missing | available |
| Specifiable precision | limited | available |
| Addition of missing features | easier | more difficult |

Table 1.1: Comparison of features present or missing in C++ and Fortran as described in [13].

# Chapter 2

# C++ Scientific Computing Techniques and Libraries

## 2.1 Expression Templates

Todd Veldhuizen who is given credit for inventing them describes expression templates in the following way [34]:

> *"Expression Templates is a C++ technique for passing expressions as function arguments. The expression can be inlined into the function body, which results in faster and more convenient code than C-style callback functions. This technique can also be used to evaluate vector and matrix expressions in a single pass without temporaries. In preliminary benchmark results, one compiler evaluates vector expressions at 95-99.5% efficiency of hand- coded C using this technique (for long vectors). The speed is 2-15 times that of a conventional C++ vector class. "*

### 2.1.1 Implementing Array Expressions Without Templates

Before explaining the mechanism of expression templates, we will discuss the problem they aim to solve first. The polymorphism features of C++ such as virtual methods and in particular operator overrides [15] give authors of scientific libraries wide range of opportunities to hide away implementation of standard arithmetic operations on complex objects such as arrays, matrices or intervals. However, the straightforward uses of these features lead to code that creates an unbearable number of temporary objects with a great performance penalty. Let us have a look at an example of such a naive implementation of classes representing two dimensional arrays and providing for element-wise addition and multiplication of them. The whole source is in appendix A, an abbreviated version of the definition of the main class and declaration of operator overrides is given in the figure 2.1.

The class stores the actual data in an allocated space pointed to by `data` and a set of operators to implement element-wise addition and multiplication of two arrays or an array and a constant. Without the utilization of templates, the functions implementing

```
class TradArray
{
private:
  // pointer to allocated memory which contains data
  double *data;

public:
  TradArray (int r0, int r1); // A constructor taking dimensions as parameters
  TradArray (const TradArray &A);  // Copy constructor
  ~TradArray (); // Destructor

  // Individual array elements are accesses in function (or Fortran) like
  // fashion.
  double &operator()(int i0, int i1);
  const double &operator()(int i0, int i1) const;

  // Assignment operator that copies the data in array a to data of this
  // object. Both this objects must have the same dimensions.
  TradArray &operator=(const TradArray &a);
  // Assignment operator that fills the entire array with the constant x.
  TradArray &operator=(const double x);
};

// Operator for adding elements of two arrays. Both arrays must have the same
// dimensions.  The result is a new temporary array containing the result.
TradArray operator+(const TradArray &l, const TradArray &r);
// The following two addition operators add constants to all elements of an
// array. Again, the result is a temporary object.
TradArray operator+(const double l, const TradArray &r);
TradArray operator+(const TradArray &l, const double r);

// The same set od operators for multiplication.
TradArray operator*(const TradArray &l, const TradArray &r);
TradArray operator*(const double l, const TradArray &r);
TradArray operator*(const TradArray &l, const double r);
```

Figure 2.1: The main class and operators of naive implementation of array operations. A few members have been omitted for the sake of brevity.

the operations have no other option but to return a new instance of `TradArray`. Executed constructors of all these temporary instances need to allocate memory to hold the results of individual operations. Moreover, when an expression is assigned to an array supposed to hold the result, it is first calculated and stored into a temporary array and only afterwards the assignment operator is invoked to copy the individual elements to the destination array.

The number of created objects can be easily measured by writing a line in each of the constructors and to the destructor and running the following code:

```
TradArray A(R1, R2), B(R1, R2), C(R1, R2), X(R1, R2);
A = 3; B = 4; C = 2;

cout << "Starting the evaluation..." << endl;

X = (A + 1) * (B + C);

cout << "Expression evaluation finished" <<  endl << endl;
```

As one would expect, such code produces the following output:

```
Ordinary constructor called
Ordinary constructor called
Ordinary constructor called
Ordinary constructor called
Starting the evaluation...
Ordinary constructor called
Ordinary constructor called
Ordinary constructor called
Destructor called
Destructor called
Destructor called
Expression evaluation finished
```

The first four constructors create objects defined by the programmer, the latter three construct results of the two additions and the multiplication. Please note that the lifespans of the temporary objects overlap and that each one of them allocates as much memory as any of the arrays programmer intended. If the sizes of the arrays are big, allocation of extra three might cause more trouble than a mere performance penalty because it can present also a memory problem, thrashing not only the processor cache but potentially even requiring more memory than the available RAM. Obviously, we would much rather have the compiler produce code equivalent to the following C code snippet. In the next section we will describe how expression templates can do just that.

```
for (i = 0; i < R1; i++)
  for (j = 0; j < R2; j++)
    X[i,j] = (A[i, j] + 1) * (B[i, j] + C[i, j]);
```

### 2.1.2 Implementing Efficient Array Expressions with Templates

In order to achieve such a radically different compiler output, the expressions must be parsed at compile time and stored as a expression type [34]. Such an expression type can then be used to evaluate the result for each of the elements separately without creating

any temporary data storage. Let us have once again look at the expression used in the last chapter:

```
X = (A + 1) * (B + C);
```

Expression templates represent this expression by a rather complicate instantiation of several templates such as the one shown in figure 2.2. The names are taken from actual type names in the example given in appendix B. `MyArrayExpr` is simply a common expression type hiding underlying expressions which can have different arities. The existence of this particular templates reduces the number of combinations operator overloading functions must be able to accept. Since all member functions of this class are inline functions [15], therefore they do not cause any performance overhead. In fact, the only member functions except the constructor this class has are different variants of `apply()` for different array ranks. All of them simply return what the underlying expression object's corresponding `apply()` function returns.

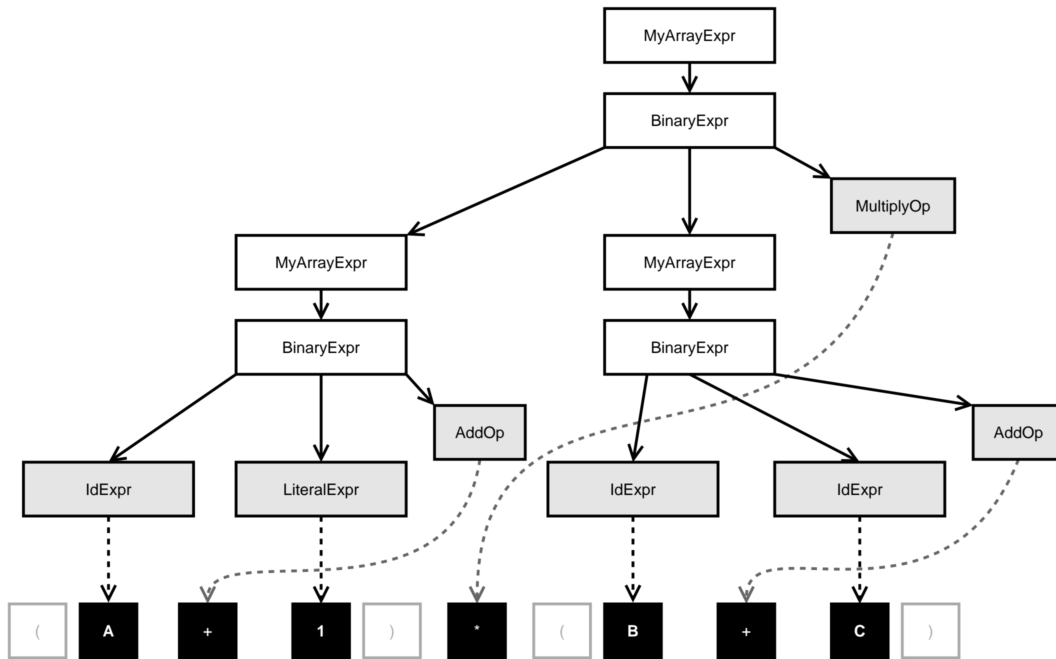



Figure 2.2: Syntax tree of the example array expression. The grey boxes represent objects which are not instantiations of a template or are instantiations of a template parametrized only by a rank (number of dimensions).

Before we start exploring the expression representation in more depth, let's have a brief look at the relevant assignment operators of the `MyArray` class. They come in three slightly different forms to provide optimal performance for arrays of different ranks, the one shown in figure 2.3 is used for two-dimensional arrays, the other two vary only in the number of for loops used. As you can see, this operator simply queries the object `expr`, which represents the expression, for values of each individual element. The expression object is capable of returning the result for one particular element alone.

The instantiations of the `BinaryExpr` template represent a binary operation on individual array elements such as addition or multiplication. The template itself has three

```
template<>
template<typename Operation>
MyArray<2> &MyArray<2>::operator= (const MyArrayExpr<Operation> &expr)
{
  int i, j;

  for (i = 0; i < dims[0]; i++)
    for (j = 0; j < dims[1]; j++)
      data[i * strides[0] + j * strides[1]] = expr.apply (i, j);

  return *this;
}
```

Figure 2.3: Array assignment operator accepting a type describing an expression.

parameters: two representing the sub-expressions that make up its operands and a small class that actually does the calculation with the given pair of `double` operands. An example of such class is `AddOp`, its entire definition is listed in figure 2.4. Other binary operations are defined by providing an equivalent class and a set of operator overrides described below. The template `UnaryExpr` is very similar to its binary counterpart, except that it requires two parameters: the operand expression and the operator, which is supposed to be a class similar to `AddOp` but with evaluate method that only has one double parameter. An even more simple type of expression is `LiteralExpr` with `apply()` methods that simply always return a single constant. Finally `IdExpr` returns elements of a given Array. It is necessary because it holds a reference to the array, unlike `MyArrayExpr` which contains the object representing its argument as a member field.

```
class AddOp
{
public:
  static double evaluate (double x, double y)
  {
    return x + y;
  }
};
```

Figure 2.4: `AddOp` – the simple class that turns generic `BinaryExpr` class into an adding machine.

It remains to be shown how such expression representation are built from the usual notation. Let us consider the addition of an array and a double constant first. The operator overriding function responsible for this particular case is the first one presented in figure 2.5. First, it creates an instance of `BinaryExpr` parametrized by `IdExpr`, `LiteralExpr` (implicitly created from `x` and `y` respectively) and `AddOp`. This new object is then encapsulated in a `MyArrayExpr` which is returned. When the `apply` method of this object is invoked, it simply calls the `apply` method of the binary expression which in turn applies the `evaluate()` method of `AddOp` on the values obtained from the instances of `IdExpr` and `LiteralExpr`. Thus the required element of the array and a constant is fetched and both are added by `AddOp`.

```
// operator+ override implementing addition of an array and a scalar
// double constant
template<int N>
MyArrayExpr<BinaryExpr<AddOp, IdExpr<N>, LiteralExpr> >
operator+ (const MyArray<N> &x, const double y)
{
  BinaryExpr<AddOp, IdExpr<N>, LiteralExpr > bin (x, y);
  MyArrayExpr<BinaryExpr<AddOp, IdExpr<N>, LiteralExpr > > r (bin);
  return r;
}
```

Figure 2.5: Two examples of operator overrides utilised in handling array expressions.

The reason why the `BinaryExpr` discussed above is encapsulated is apparent if we look at the operator override implementing a binary operation on top of two complex expressions such as the multiplication in the example given on page 11. The code of the considered function is listed in figure 2.6 and closely resembles the `operator+` discussed earlier. The key difference is that the instance of `BinaryExpr` is parametrized by two `MyAddrExprs` which may represent any expression, whether it is binary, unary, or completely different. This significantly reduces the number of required variants of operator overloading functions. The appendix B defines these operators for addition and multiplication for the following combinations of operands:

- two complex expressions represented by `MyAddrExpr`,
- a complex expression and an array and the other way round,
- a complex expression and a `double` constant and the opposite order of the two,
- an array and a `double` constant and the other way round, and
- two arrays.

### 2.1.3 Efficiency of the Resultant Code

As far as the efficiency of compiler output is concerned, intermediate debugging outputs from compilers such as the one given in appendix C prove that the resultant code is indeed very similar to the C code given in the end of section 2.1.1. In particular, all the overhead caused by the number of functions involved can be easily inlined away. Performance measurements of expression templates can be found for example in [34] and [36]. Both of these sources claim they can achieve performance on par with optimized Fortran 77 code. Performance measurements of any of the libraries discussed in the end of this chapter (such as [33] or [25]) are also relevant because all of them are heavily based on expression templates. Understanding how expression templates work and what code they produce, especially the rather complex types of call graphs and data structures they use, is therefore vital for designing compiler optimizations aimed at scientific computations implemented in C++.

```
// operator* override handling addition of two compound expressions
template<typename A, typename B>
MyArrayExpr<BinaryExpr<MultiplyOp, MyArrayExpr<A>, MyArrayExpr<B> > >
operator* (const MyArrayExpr<A> &x, const MyArrayExpr<B> &y)
{
  BinaryExpr<MultiplyOp, MyArrayExpr<A>, MyArrayExpr<B> > bin(x, y);
  MyArrayExpr<BinaryExpr<MultiplyOp, MyArrayExpr<A>, MyArrayExpr<B> > > r(bin);
  return r;
}
```

Figure 2.6: Two examples of operator overrides utilised in handling array expressions.

### 2.1.4 Other Uses of Expression Templates

Just as the quotation in the beginning of this chapter says, expression templates can be used anywhere an expression should be passed as a function argument with minimum runtime overhead. The original paper by T. L. Veldhuizen [34] describes an implementation of a function evaluating a given expression at a range of points and suggests a number of other uses.

## 2.2 Language Extensions

Even though compilers nowadays often utilize aggressive inlining strategies (see section 3.1), Richard Günther [21] discovered that the inlining startegy of GCC 4.0 sometimes fails to carry out inlining necessary to remove the extra level of abstraction and complex call relationships very complex expression templates introduce. He solved this problem by introducing a special `flatten`[1] function attribute to hint the compiler at completely inlining all calls inside an expression template expansion. The attribute effectively turns the function into a leaf node in the programs call graph. The programmer is responsible for avoiding any recursion in the function itself and those it calls.

The C99 [16] standard introduced a `restrict` pointer qualifier. Marking a pointer with it informs the compiler that the object this pointer refers to is accessed exclusively with this pointer or expressions based on it and there are no other means of accessing it, whether direct or indirect. This qualifier therefore allows the programmer to explicitly override the relaxed aliasing rules described in chapter 1. Even though this qualifier is not part of any C++ standard, many compilers of this language implement it too [9, 3, 24]. To quote a quick example from the GCC manual [3], in the body of the following function, “`rptr` points to an unaliased integer and `rref` refers to a (different) unaliased integer.”

```
void fn (int *__restrict__ rptr, int &__restrict__ rref)
{
  /* ... */
}
```

Finally, some researchers decided to go further than exploiting the current language features or adding new ones and decided to developed source-to-source transformation tools such as Sage++ [11]. However, these tools are outside of the scope of this work.

[1] Called *leafify* in the original proposal.

## 2.3 C++ Libraries for Scientific Computing

### 2.3.1 Blitz++

Blitz++ [33] is a C++ library for scientific computing. In particular it provides dense numerical arrays similar to those in Fortran 90 and heavily utilizes expression templates (see section 2.1.2) and template metaprograms [35] to achieve high performance array manipulations. Blitz++ arrays can have various ranks and elements of complex nature, they are reference counted and support wide range of operations and notations.

Moreover, the library itself attempts to perform the following optimizations:

- loop interchange and reversal,
- hoisting stride calculations,
- collapsing inner loops,
- partial unrolling,
- common stride optimizations, and
- tiling.

### 2.3.2 Parallel Object Oriented Methods and Applications (POOMA)

POOMA [32, 29] framework grew out of the Object-Oriented Particle Simulation class library which was designed specifically for particle-in-cell simulations. Both were developed at Los Alamos National Laboratory. Applications of POOMA include Numerical Tokamak, molecular dynamics, high speed multimaterial CFD and rhenological flow simulations. POOMA was also used in Richard Günther's work on Three-dimensional Parallel Hydrodynamics and Astrophysical Applications which is often referred to as Tramp 3D [21].

The main goal of POOMA framework was to allow writing of portable code which would run on various platforms including serial, distributed and parallel computers while retaining high degree of reusability and efficiency of the generated code. It has a component architecture which simplifies rapid application development. Users of POOMA can develop their code on a serial machine and then run it without any change on a parallel or a distributed computer. Even though the authors themselves admit that gains on serial architectures are minimal, POOMA applications still present valuable benchmarks because they also use expression templates (see section 2.1.2), even though they are more complex to allow for parallelism[2].

[2]POOMA developers call them *chained expression objects*.

# Chapter 3

# Existing Optimizing Techniques

This chapter discusses several well-known optimizing techniques that contemporary compilers exploit in order to boost performance of C++ scientific applications. Even though a huge number of different optimizations have been proposed and implemented in the past, we will concentrate on the best known ways of removing the abstraction overhead of expression templates that GCC and others utilize and on mechanisms of special relevance to the new analysis and transformations proposed in chapter 4. Finally, we will briefly describe loop optimization techniques which are usually credited most with increase of performance of scientific computations.

## 3.1 Procedure integration

Procedure integration or inline substitution [30] replaces calls to functions with copies of their bodies. This not only eliminates the call overhead but perhaps more importantly allows other optimizations to take place, especially if the replaced call site was inside a loop. It also provides a much bigger scope for subsequent intraprocedural optimizations which greatly increases their potential and for scheduling and register allocation [6]. The C++ language allows the users to mark functions they would like to have inlined wherever possible either by explicitly using a function specifier or by defining it within a class definition [15][1]. Furthermore, GCC can perform automatic inlining at call sites where it deems it beneficial [4].

When deciding which calls should be substituted with function bodies, GCC keeps a work list sorted according to *badness* of individual call sites. The smaller the badness, the better chances of the associated candidate to be inlined. The badness is calculated from estimated code growth incurred by inlining and measured or guessed *frequency* with which this call is executed. The frequency is either based on profiling information if that is available or estimated from the number of loops this call is nested in [23, 8]. The potential candidates are also constrained by a number of limits such as maximum growth of functions and the whole compile unit. The algorithm performing the inlining is then a simple greedy one: the candidate call sites with the smallest badness are selected and marked for inlining one after another as long as there are any. Each time

[1] GCC substitutes calls to member functions defined within a class definition only at optimization levels -O1 and higher. See [3] for details.

such an inlining decision is made, the call graph and badnesses of a number of functions must be recomputed. Additionally, GCC also performs separate early inlining of functions which are small enough that the cost of running them is smaller than the cost of executing the call [23, 4]. This optimization is not a true interprocedural pass, it simply processes functions in topological order and inlines all call sites which satisfy the condition above. Other production compilers adopt similar strategies. For example, the HPUX compiler [14] also uses a greedy algorithm and calculates *goodness* of call sites from a wide range of values including expected effects of enabled optimization savings and cache effects.

The GCC version 4.3.0 which is currently under development performs inlining of functions which have already been converted to SSA [23, 4, 30] and after a number of early intraprocedural optimizations (such as constant propagation, dead code elimination and others) have taken place. These optimized functions are inlined into those who have not been transformed yet. This helps to better estimate the effects of inlining on the final code size, avoids multiple transformations of the same code, and saves memory and time during compilation [23].

In order to evaluate the effects of the two types of inlining, we ran two series of experiments, one with early inlining enabled and the other without it, in which we gradually relaxed the inlining limits and observed the effect on execution and compile time of the Tramp 3D benchmark [21]. We carried out the measurements twice, the difference in between all corresponding values were within 1% range. The computer used for the measurements was AMD Athlon 64 X2 Dual Core Processor 3800+ equipped with 2 GB of RAM. The presented times are those the compiler and the benchmark spent in their user space, i.e. excluding operating system overhead. We used GCC 4.3 which is currently under development, specifically the one with the subversion revision number 123774 without any modifications whatsoever.

The options used in all compilation were:

```
g++ -O2 -funroll-loops -ffast-math -fno-early-inlining -finline-functions tramp3d-v4.cpp
```

Moreover, the parameter max-inline-insns-auto was set to either 90 (the default) or 180 and parameters large-function-growth and inline-unit-growth varied from 20 to 400 and 6 to 120 respectively. The switch -fno-early-inlining was used to prevent early inlining in the appropriate series of measurements. When measuring the execution time, we launched Tramp 3D with the following arguments:

```
./tramp3d –cartvis 1.0 0.0 –rhomin 1e-8 -n 40
```

The results are presented in three graphs in this section and in tables 3.1. and 3.2. The x-axis in the graphs describes how many times the inlining limits determining the maximum growth of a large function and of a large compilation unit were relaxed[2]. 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 uses the POOMA [29] library and so exploits expression templates a lot. Consequently it consists of many tiny functions which are

[2] GCC documentation [3] states that by default, functions deemed large can grow to twice the original size and large compilation units can grow 1.5 times. Large functions are those having more than 2700 instructions in the internal gcc interpretation and large units have more than 10000.

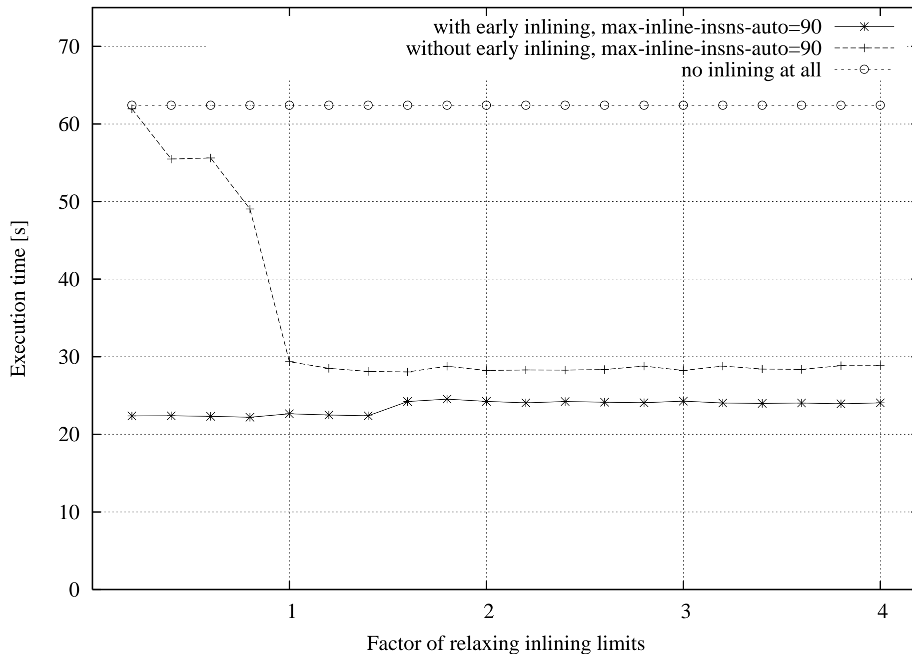



Figure 3.1: Effect of inlining on execution time of Tramp 3D.

near to impossible to optimize on their own and thus the expected effect of inlining on the execution time of the benchmark was big. The results confirmed it. The graph in figure 3.1 shows the impact regular inlining had on Tramp 3D execution time with and without early inlining and also the execution time of the benchmark when no inlining took place at all . As you can see, the non-inlined executable is almost three times slower than most of the versions produced while inlining. In this particular benchmark, the early inlining technique proved to be more effective than traditional inlining with any given parameters, even though when the limits were relaxed it was able to achieve similar results. Another important observation is that very relaxed inlining limits do not buy any performance increase from some point on and in fact can degrade it a little (see figure 3.2).

The drawbacks of procedure integration all stem from the fact that both the overall size of code and size of individual functions significantly increase. Apart from the nuisance of ending up with bigger executables, both increases have two indirect and more unpleasant consequences. First, the bigger the code size, the greater the chance of thrashing the processor instruction cache which is growing more and more costly as the performance gap between processors and main memory widens. This is probably the cause of the small performance drop exhibited in figure 3.2. Second, by producing excessive code, inlining slows down all the different types of analysis and code transformation that the compiler carries out after inlining (see figure 3.3) and makes them use more memory (see tables 3.1 and 3.2). To make matters worse, some of these passes aren't linear in either time or space and so providing them with huge functions might have a disastrous impact on memory requirements or the compile time.

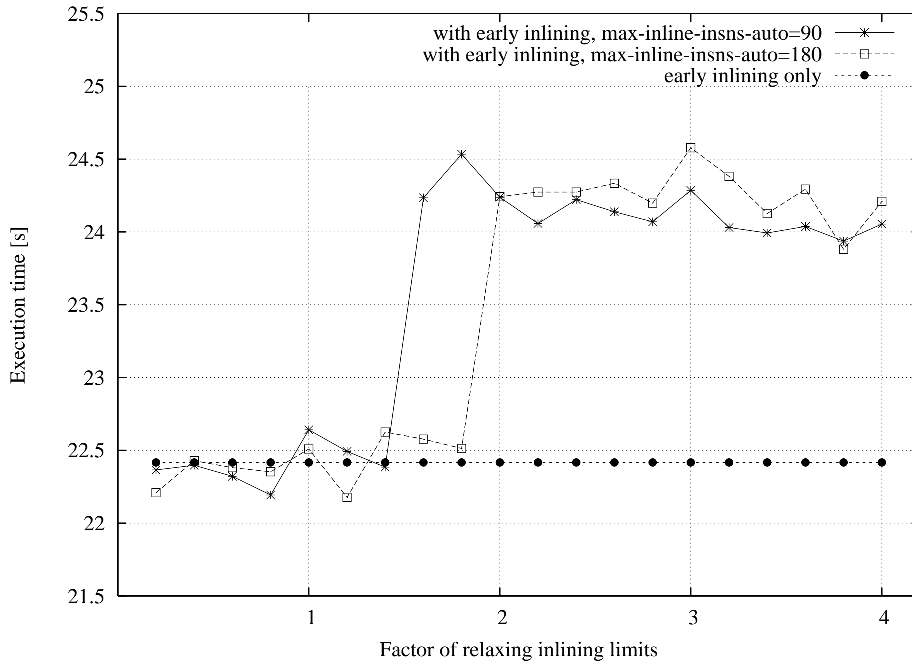



Figure 3.2: Effect of both early inlining and traditional inlining on execution time of Tramp 3D.

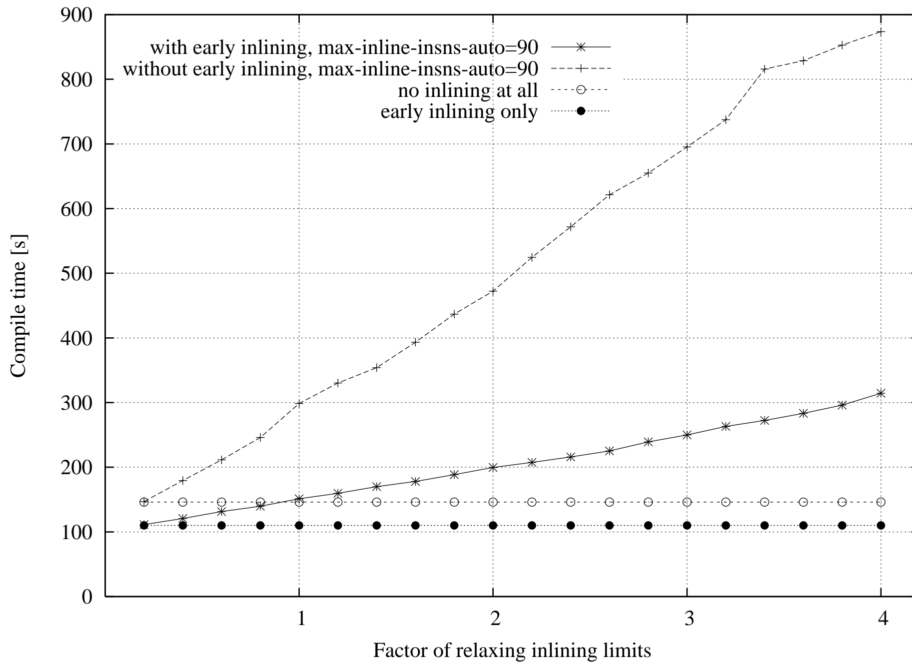



Figure 3.3: Effect of inlining on compilation time of Tramp 3D.

| Max Func Size | Relaxation Factor | Func Growth | Obj Growth | Compile Time | Exec Time | RAM Req |
|---|---|---|---|---|---|---|
| 90 | 0.2 | 20 | 6 | 111.327 | 22.365 | 164M |
| 90 | 0.4 | 40 | 12 | 120.660 | 22.397 | 164M |
| 90 | 0.6 | 60 | 18 | 131.472 | 22.321 | 164M |
| 90 | 0.8 | 80 | 24 | 139.721 | 22.193 | 165M |
| 90 | 1.0 | 100 | 30 | 151.329 | 22.641 | 163M |
| 90 | 1.2 | 120 | 36 | 159.686 | 22.493 | 158M |
| 90 | 1.4 | 140 | 42 | 169.931 | 22.385 | 165M |
| 90 | 1.6 | 160 | 48 | 178.099 | 24.234 | 166M |
| 90 | 1.8 | 180 | 54 | 188.632 | 24.534 | 166M |
| 90 | 2.0 | 200 | 60 | 199.556 | 24.238 | 167M |
| 90 | 2.2 | 220 | 66 | 207.497 | 24.058 | 167M |
| 90 | 2.4 | 240 | 72 | 216.018 | 24.222 | 167M |
| 90 | 2.6 | 260 | 78 | 225.162 | 24.138 | 167M |
| 90 | 2.8 | 280 | 84 | 239.123 | 24.070 | 168M |
| 90 | 3.0 | 300 | 90 | 250.008 | 24.286 | 169M |
| 90 | 3.2 | 320 | 96 | 263.120 | 24.030 | 157M |
| 90 | 3.4 | 340 | 102 | 272.413 | 23.993 | 171M |
| 90 | 3.6 | 360 | 108 | 283.386 | 24.038 | 170M |
| 90 | 3.8 | 380 | 114 | 296.163 | 23.937 | 171M |
| 90 | 4.0 | 400 | 120 | 314.440 | 24.054 | 174M |
| 180 | 0.2 | 20 | 6 | 111.583 | 22.209 | 162M |
| 180 | 0.4 | 40 | 12 | 120.508 | 22.429 | 164M |
| 180 | 0.6 | 60 | 18 | 129.988 | 22.381 | 164M |
| 180 | 0.8 | 80 | 24 | 139.773 | 22.353 | 164M |
| 180 | 1.0 | 100 | 30 | 152.034 | 22.509 | 157M |
| 180 | 1.2 | 120 | 36 | 162.810 | 22.177 | 155M |
| 180 | 1.4 | 140 | 42 | 172.239 | 22.625 | 157M |
| 180 | 1.6 | 160 | 48 | 181.127 | 22.577 | 166M |
| 180 | 1.8 | 180 | 54 | 188.684 | 22.513 | 167M |
| 180 | 2.0 | 200 | 60 | 198.472 | 24.242 | 167M |
| 180 | 2.2 | 220 | 66 | 207.745 | 24.274 | 159M |
| 180 | 2.4 | 240 | 72 | 220.378 | 24.274 | 167M |
| 180 | 2.6 | 260 | 78 | 229.322 | 24.334 | 168M |
| 180 | 2.8 | 280 | 84 | 237.647 | 24.198 | 171M |
| 180 | 3.0 | 300 | 90 | 247.147 | 24.578 | 168M |
| 180 | 3.2 | 320 | 96 | 261.452 | 24.382 | 169M |
| 180 | 3.4 | 340 | 102 | 273.317 | 24.126 | 171M |
| 180 | 3.6 | 360 | 108 | 283.922 | 24.294 | 170M |
| 180 | 3.8 | 380 | 114 | 295.322 | 23.881 | 170M |
| 180 | 4.0 | 400 | 120 | 308.099 | 24.210 | 170M |

Table 3.1: All results of measurements of inlining effects on Tramp 3D. The series with early inlining allowed.

| Max Func Size | Relaxation Factor | Func Growth | Obj Growth | Compile Time | Exec Time | RAM Req |
|---|---|---|---|---|---|---|
| 90 | 0.2 | 20 | 6 | 146.977 | 61.956 | 156M |
| 90 | 0.4 | 40 | 12 | 179.567 | 55.475 | 157M |
| 90 | 0.6 | 60 | 18 | 211.453 | 55.615 | 158M |
| 90 | 0.8 | 80 | 24 | 245.923 | 49.047 | 156M |
| 90 | 1.0 | 100 | 30 | 298.623 | 29.366 | 160M |
| 90 | 1.2 | 120 | 36 | 330.389 | 28.510 | 158M |
| 90 | 1.4 | 140 | 42 | 353.914 | 28.114 | 159M |
| 90 | 1.6 | 160 | 48 | 393.413 | 28.042 | 158M |
| 90 | 1.8 | 180 | 54 | 436.743 | 28.778 | 162M |
| 90 | 2.0 | 200 | 60 | 472.278 | 28.222 | 159M |
| 90 | 2.2 | 220 | 66 | 524.633 | 28.290 | 165M |
| 90 | 2.4 | 240 | 72 | 571.864 | 28.282 | 164M |
| 90 | 2.6 | 260 | 78 | 621.511 | 28.346 | 165M |
| 90 | 2.8 | 280 | 84 | 655.053 | 28.790 | 161M |
| 90 | 3.0 | 300 | 90 | 695.315 | 28.230 | 166M |
| 90 | 3.2 | 320 | 96 | 737.478 | 28.794 | 162M |
| 90 | 3.4 | 340 | 102 | 815.775 | 28.422 | 168M |
| 90 | 3.6 | 360 | 108 | 828.636 | 28.370 | 173M |
| 90 | 3.8 | 380 | 114 | 852.777 | 28.850 | 174M |
| 90 | 4.0 | 400 | 120 | 873.563 | 28.842 | 171M |
| 180 | 0.2 | 20 | 6 | 146.569 | 64.428 | 157M |
| 180 | 0.4 | 40 | 12 | 179.231 | 57.404 | 158M |
| 180 | 0.6 | 60 | 18 | 206.777 | 55.951 | 158M |
| 180 | 0.8 | 80 | 24 | 243.143 | 52.223 | 159M |
| 180 | 1.0 | 100 | 30 | 287.038 | 40.543 | 156M |
| 180 | 1.2 | 120 | 36 | 331.681 | 28.190 | 161M |
| 180 | 1.4 | 140 | 42 | 361.751 | 28.598 | 161M |
| 180 | 1.6 | 160 | 48 | 388.756 | 28.566 | 162M |
| 180 | 1.8 | 180 | 54 | 430.655 | 28.186 | 162M |
| 180 | 2.0 | 200 | 60 | 466.749 | 28.562 | 160M |
| 180 | 2.2 | 220 | 66 | 510.452 | 28.262 | 165M |
| 180 | 2.4 | 240 | 72 | 569.940 | 28.130 | 168M |
| 180 | 2.6 | 260 | 78 | 620.295 | 28.530 | 166M |
| 180 | 2.8 | 280 | 84 | 654.101 | 28.598 | 163M |
| 180 | 3.0 | 300 | 90 | 734.286 | 28.286 | 172M |
| 180 | 3.2 | 320 | 96 | 762.492 | 28.258 | 163M |
| 180 | 3.4 | 340 | 102 | 788.485 | 28.178 | 164M |
| 180 | 3.6 | 360 | 108 | 822.759 | 28.558 | 164M |
| 180 | 3.8 | 380 | 114 | 849.513 | 28.370 | 164M |
| 180 | 4.0 | 400 | 120 | 872.147 | 28.198 | 164M |

Table 3.2: All results of measurements of inlining effects on Tramp 3D. The series with early inlining suppressed.

Unlike in Fortran, since formal parameters of a function in C++ may be aliases of each other, there are no issues with aliasing information getting lost during inline substitution similiar to those described in [18].

## 3.2 Interprocedural Constant Propagation

Interprocedural analysis and optimizations are often considered an alternative to procedure integration because presumably they tend to be less costly in terms of code size growth. On the other hand, modifying most well-known intraprocedural optimizations to work interprocedurally is very difficult. Moreover, interprocedural code transformations also very often require substantial code expansion in the caller, callee or both so that they are safe [6].

Fortunately, interprocedural constant propagation [12] is one of interprocedural optimization techniques that does not suffer any of the two drawbacks and is implemented in GCC. The analysis aims to associate each formal parameter of each function with an element of a lattice depending on whether the actual arguments used when calling this function are known to be the same constant (represented by the constant) or whether they vary or the compiler cannot reason about their value (both cases are represent by the lattice element *bottom*)[3]. It proceeds in three main stages:

1. *Intraprocedural analysis.* During this phase, the compiler examines bodies of individual functions, one at a time, and computes a *jump function* for each actual argument at each call site. The jump function specifies either that the actual parameter is always a constant, is never a constant, its constantness cannot be determined or how its value can be calculated from the formal parameters of the calling function or even return values of other functions it calls.

   The interprocedural constant propagation implemented in GCC utilizes simple jump function that denotes an actual argument is a known constant, an unknown or the value passed to a formal parameter of the caller. It is the *pass through* type of jump functions discussed in the Callahan's paper [12] and which is recommended as the most cost-effective in [20].

2. *Interprocedural analysis.* Given the jump functions, the compiler performs the Wegman–Zadeck [37] method for propagating constants within a single function. In principle, the compiler applies the meet lattice operation to all corresponding arguments with which a function is called and these values are propagated through the call functions until all the lattices stabilize.

3. *Substitution.* Finally, all formal parameters that have been proved constant are initialized with the constant value at the beginning of the function. The forward copy propagation pass will move its value to the uses later on.

GCC interprocedural constant propagation is capable of analyzing and propagating simple integers, floating point numbers and pointers. It does not attempt to analyze aggregate data types and thus is not capable of propagating information stored within objects. This optimization pass also analyzes and manipulates function in the SSA form

[3] The *top* element is used during the analysis to denote that the value of the parameter has not been fully determined yet.

which simplifies construction of jump functions which is equivalent to simple query for ssa name definition.

Some measurements of the behaviour and effects of this pass when compiling a few examples expression template-based scientific code are given in table 3.3. The number of propagated constants were never big and the impact on the execution time of the application were negligible.

| **Benchmark** | **Execution time without IPA-CP** | **Replaced parameters** | **Execution time with IPA-CP** | **Improvement** |
|---|---|---|---|---|
| Tramp 3D | 22.613s | 46 | 22.433s | 0.8% |
| Blitz++ Acoustic 3D | 49.803s | 6 | 50.099s | -0.6% |
| Blitz++ Acoustic 2D | 4.440s | 1 | 4.452 | -0.3% |

Table 3.3: Measured effects of interprodecural constant propagation (IPA-CP).

## 3.3 Procedure cloning

Procedure cloning, or procedure specialization [30] means creating a special copy of a function for call sites which are known to use the same set of constant parameters. The data flow analysis this interprocedural optimization bases its decisions on is very much like the one utilized by interprocedural constant propagation, except that the interesting lattices are associated with actual arguments rather than with formal parameters. Nevertheless, the decision whether cloning at a particular call site should or should not be carried out is not a trivial one. Only some opportunities for cloning actually result in faster code, mainly by allowing subsequent intraprocedural optimizations. Moreover, unnecessary cloning always leads to growth of code which can cause problems discussed in section 3.1 and is potentially exponential (see figure 3.4). Therefore, Cooper, Hall and Kennedy [17] suggested that cloning is carried out only when it exposed constant that:

- specifies a dimension of an array,
- determines control flow, or
- appear in a subscript expression.

Some compilers use a greedy algorithm similar to the one described in section 3.1 and perform cloning as long as there opportunities within certain limits. This led Das to propose doing both at once [19].

GCC currently does not clone functions. The interprocedural constant propagation pass discussed in the previous section makes a copy of each function it transforms but that happens only if a formal parameter is the same constant in all calls to the function [23, 4]. On the other hand, true cloning as described above should be capable of creating a number of specializations of a given function.

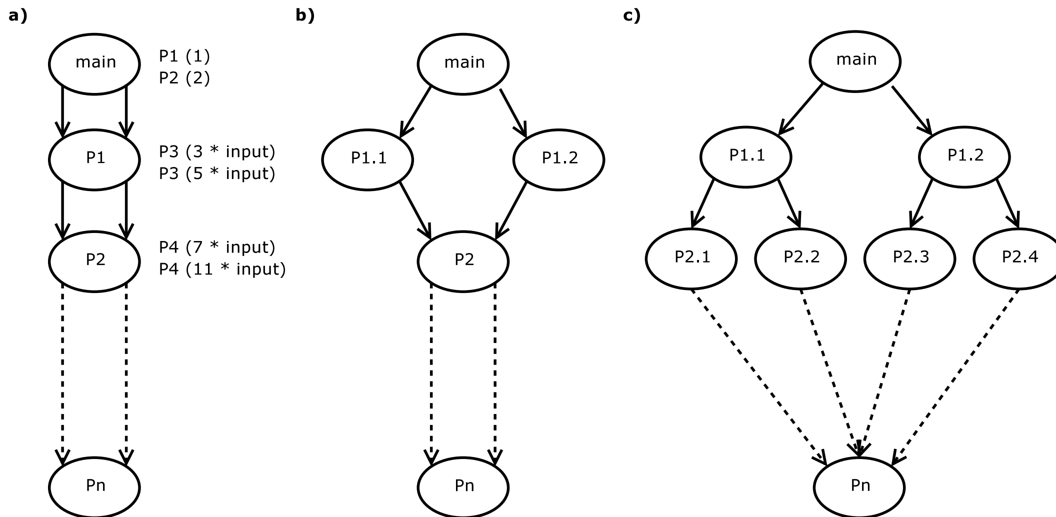



Figure 3.4: Cloning can lead to exponential compilation unit growth [17]. **a)** depicts the initial situation, **b)** the effect of cloning function P2, and **c)** what happens after cloning P3.

## 3.4 Scalar Replacement of Aggregates

Because most optimization techniques do not consider components of aggregates such as arrays, structures or objects, GCC incorporates a technique to replace the aggregate with its components. This technique is called *scalar reduction of aggregates* (SRA) [30]. Once the replacement has taken place, the newly independent scalars can be subject to constant and copy propagation, register allocation and so on. GCC scalar replacement pass proceeds in three stages [4]:

1. The pass identifies candidates for replacement. These must be variables of scalar type that cna be proved not to be aliased in any way. SRA ensures they don't by refusing to operate on any aggregate that must reside in memory, i.e. which has its address taken, which is a global variable and so on. The pass also excludes any variables marked as volatile [15] and variables that are hard to decompose such as unions or structures of variable size.

2. The pass scans the current function and analyzes how the identified candidates are used. In particular, it determines the number of times the candidate and its member are needed as a part of the whole aggregate and the number of times the members and whole aggregates are copied.

3. SRA uses the statistics from the previous step to select the candidates which are to be replaced and instantiates the replacement variables. Basically, a scalar member is replaced by a separate variable when it is used individually more frequently than as a part of the aggregate.

4. Finally, the pass traverses the whole function once again, replacing any uses of the old candidates with the new variables.

Heeding the requirements given in the first point above, SRA cannot substitute any aggregates which had their address taken. This also applies for example to those which have been passed by reference to another function and that also applies to `this` pointers when calling an object's method. Nevertheless, these cases are often removed by inline substitution (see section 3.1), whether requested by the programmer or performed automatically by the compiler. In order to demonstrate this, we have carried out the following set of experiments, similar to a one presented in section 3.1. We have run the compilation of the Tramp 3D benchmark [21] with disabled early inlining and allowed ordinary inlining and with the same set of options as before, tightening the inlining growth limits by the factors of 1, 2, 3, 4 and 5. The settings correspond to the steeply decreasing line in figure 3.1 and relaxing factors 1/5, 2/5, 3/5, 4/5 and 1. This time, however, we have slightly modified the compiler the SRA pass to dump various information about its activity. The number of aggregate members turned into separate scalar variables in each of these experiments is shown in figure 3.5 and, as you can see, they significantly increase as more call sites are inlined. On the other hand, the number of aggregates which were local variables but were not considered because they needed to live in memory also increased from nearly forty thousand to almost forty-five thousand[4]. This was the motivation to design and implementation of the analysis described in chapter 4.

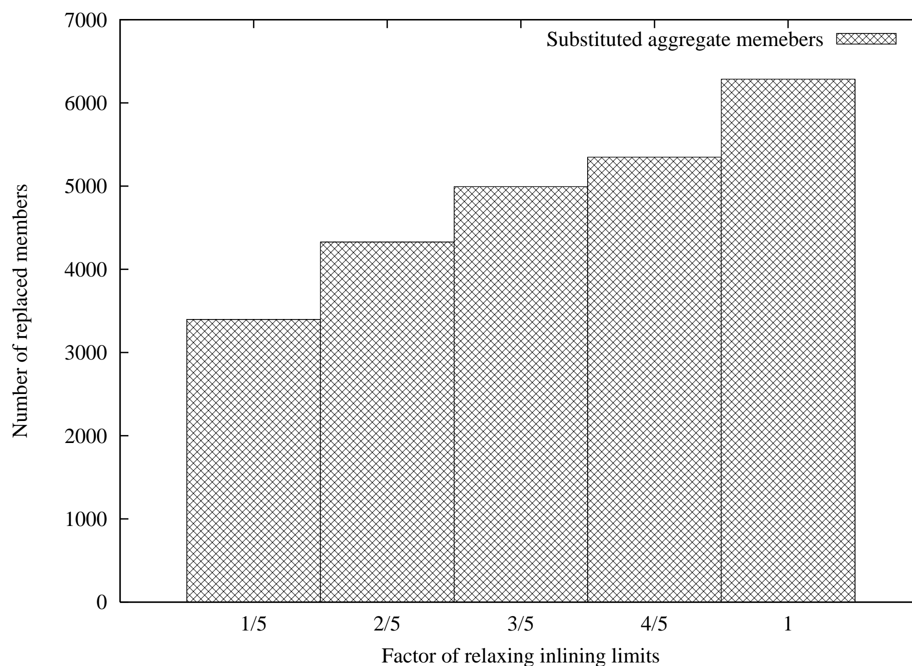



Figure 3.5: Effect of inlining on the number of instantiated scalar replacements of aggregate members when compiling Tramp 3D benchmark.

[4]Please note that the SRA pass is run twice during compilation and so some of these reported candidates have been refused (and included in the statistics) twice. This does not apply to data in the graph since each member was replaced only once.

## 3.5 Loop transformations

Scientific applications spend most of their time in loops and thus the primary emphasis when compiling such programs is always on loop transformations [7]. Most of these loop transformations are dependant on *dependence analysis* [26] declaring them safe. Loop transformations which are implemented in GCC [4] and which use dependency analysis include:

- *Loop interchange*, which can exchange the position of two loops in a perfect loop nest [7],
- *loop reversal* that changes the direction in which a loop traverses its dimension of the iteration space [7],
- *loop skewing*, which traverses the iteration space in a diagonal manner [7],
- *loop scaling*, which corresponds to replacing a loop iteration variable with an integer multiple of it [28],
- *strip mining*, which changes a granularity of an operation to enhance potential for parallelism [7], and
- *vectorization* [7] that attempts to convert loops into vector instructions of the target architecture.

Moreover, many other have been proposed and might be implemented one day. Dependence within loops is usually captured by construction of distance vectors [26, 7] which describe which iterations – vectors from the iteration space – access the same array elements and thus their order of execution must be preserved. Nevertheless, distance vectors can be constructed and are often useful only when the array access strides are known at compile time. C++ scientific applications, however, often have strides encapsulated within classes and so even when they are constant at compile time, these constants are not propagated to the places where they are needed most.

Additionally, there are loop transformations that are not based on dependence analysis but can only be performed when the number of iterations is known at compile time. Others do not require it but be performed a lot better when it is so Examples of such optimizations include induction variable optimizations [30], loop unrolling, loop peeling, loop spreading [7], and many others. Like in the case of strides, array and thus iteration space dimensions are often encapsulated within a class and often no contemporary analysis can extract it.

# Chapter 4

# Interprocedural Analysis of Aggregates

The objects representing various scientific and mathematical entities in C++ code often contain constants and information that would be invaluable for various kinds of intraprocedural analysis such as dependence analysis [26] and loop transformations. For example, array objects in both Blitz++ and the example in appendix B contain the array sizes and strides for each of their dimensions. Yet the only contemporary way of extracting these information is the scalar replacement of aggregates discussed in section 3.4 which is incapable of decomposing objects used by a number of functions. Nevertheless, the objects often never leave the current compilation unit and are rarely passed to library functions[1] and thus are entirely under the control of the compiler.

The *interprocedural analysis of aggregates* proposed in this chapter aims to allow extraction of such information from well-behaving objects. It tracks how objects are created and passed around in between functions of the current compilation unit and creates data structures along call graph nodes and edges that capture this behaviour. It also detects uses which may cause uncontrollable aliasing of these objects and interprocedurally propagates this information to all relevant functions. 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.

The pass does not intend to be able to comprehend complex algorithms or data structures. It primarily aims to be able to prove a structure passed around as the `this` pointer in between different methods of a class is not aliased in any way and possibly uncover various properties of their members, such as the constantness. In general, we are interested in identifying aggregates that are allocated either automatically or dynamically, are kept in simple local variables only and are passed to other known and well-behaving functions. Having done that, we can do data flow analysis like constant propagation. The pass 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.

[1] Except for dynamic memory allocation deallocation which can be treated a special case.

3. *Interprocedural usability propagation.* The compiler 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.

5. *Jump and return function building stage.* This intraprocedural phase is the first part of the constant propagation. It calculates jump functions describing actual arguments of each call site within a function and builds the return function of the each procedure.

6. *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.

7. *Replacement stage.* The compiler replaces uses of any member which was discovered to be constant with the appropriate constant.

8. *Cleanup.* Finally, the pass frees all memory it no longer uses.

## 4.1 Core data structures

The three most important data structure types at the heart of interprocedural analysis of aggregates are: `struct node_info` describing functions, `struct ocp_item` representing aggregates, and `struct edge_info` characterizing call sites. Structures concerning information associated with call graph edges are created in the second stage as individual call sites are encountered. The first two are created and initialized right after the pass gains control. Before describing how this is done, we will have a look at the data structures themselves.

### 4.1.1 Data associated with aggregates

The analysis examines structures, arrays and standalone pointers to structures and arrays. It represents these aggregates and their members by a single structure called `ocp_item` (see figure 4.1). From now on, an *item* always means an instance of this structure. Naturally, an item contains references to the internal representation of the program, such as the function this item belongs to (`node`), its declaration (`decl`) and SSA name (`name`) of pointers. Because an item can find itself in two separate queues at the same time, the structure has two "next" pointers (`fwd` and `bck`). This structure also directly contains the return function and initial value calculated by the constant propagation part of the pass. The other fields may need a bit more thorough explanation.

We have already mentioned the same structure represents both whole aggregates and their members. The way it works is depicted in figure 4.2. There is an array of `struct ocp_items` allocated for all fields of a particular structure and its address is stored in the `children` pointer of the item representing the structure. The number of these children is available in field `children_count`. Because the algorithms involved in the pass often need to examine the out-most item containing a given item, it is always directly

```
struct ocp_item
{
  /* The function to which this item belongs, it is NOT updated when the node
     is cloned and therefore should not be used during the replacement stage.*/
  struct cgraph_node *node;

  /* Declaration this item refers to */
  tree decl;

  /* When the item describes a particular ssa_name, this is it */
  tree name;

  /* If this SSA_NAME points to an object that is described by a different
     item or even a part of a different item, target points to the real thing.
     If non-NULL, the item it points to should be used for all operations
     (except some cases of ssa_name item queries). */
  struct ocp_item *target;

  /* Pointer to the top level item representing the object in which this item
     resides. Points to this item if it is itself the top level one. */
  struct ocp_item *object;

  /* Number of children of this item */
  int children_count;
  /* Vector containing elements refering to individula fields of a record: */
  struct ocp_item *children;

  /* Bits to set and clear, see commentd on struct ocp_item_flags.  */
  struct ocp_item_flags flags;

  /* Forward queue binder for interprocedural analysis. It is also used to
     chain together array type items within a function through NEXT_ARRAY. */
  struct ocp_item *fwd;
  /* Backward queue binder for interprocedural analysis */
  struct ocp_item *bck;

  /* Index of appropriate lattice values in temporary arrays */
  int index;
  /* Number of consumed indices by this item */
  int size;

  /* Initial value of a parameter.  Determined by interprocedural constant
     propagation phase. */
  struct lattice_item init;

  /* Return function of a parameter. Determined by jump and return building
     stage. */
  struct lattice_item retf;
};
```

Figure 4.1: The data structure describing aggregates and their members.

accessible through the `object` field. The out-most items are also called *top level items* and their `objects` point to them. Conversely, items that do not have any children are referred to as *leaf items*. Items representing arrays are always leaf items.

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

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

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

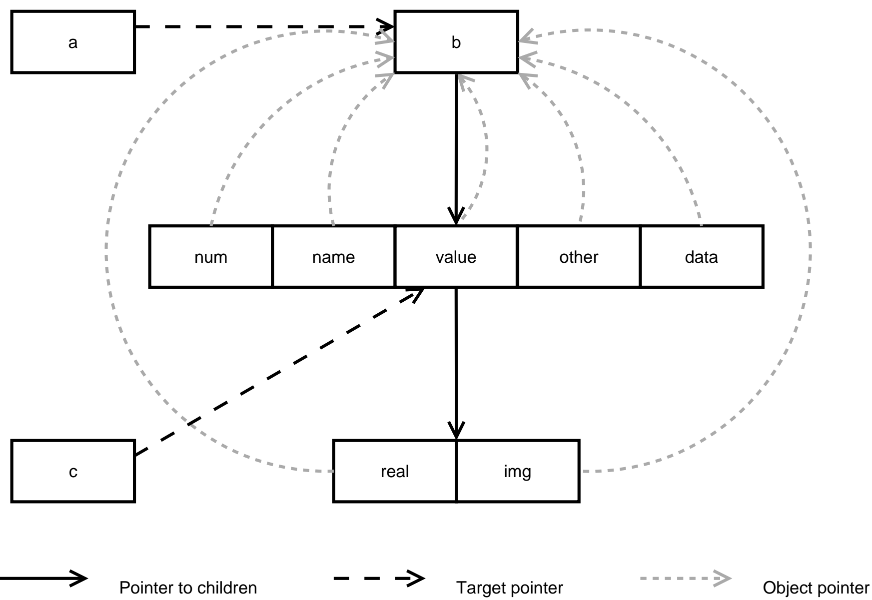



Figure 4.2: An example aggregate variables and their representation as items. The figure also demonstrates the meaning of `children`, `object` and `target` fields of an item.

The same aggregate structures can be accessed through different pointers. Many pointers refer only to a part of another object, in particular, pointers to an object's ancestors do. In order to maintain this information, items have the `target` field. This field is meaningful only in top-level items and if it is non-NULL, it points to the item corresponding to the object which is actually accessed through the pointer. Targets are not transitive, they always point to the item that actually represents an object[2]. When a target is set, most of

[2]That means `(!item.target || !item.target->target)` always holds true.

the other fields of an item are meaningless and undefined. Nevertheless, there are items corresponding to pointers which do not have their targets set and represent an entity of their own, such as those referring to dynamically allocated objects and those representing formal parameters.

The `flags` field contains various bits that describe the associated object:

**unusable** – This flag has different meaning for top-level items and those at lower levels. When set in a top-level item, the corresponding object for some reason cannot be reliably analysed, usually because its address escapes uncontrollably. On the other hand, the object's behaviour can be worth analyzing if its `backward_unusable` flag is not set (see section 4.4).

When set in an item representing a member of a structure, the given item alone is somehow unfit to be analyzed, usually because it has an unsupported type.

**forward_unusable** – Set when the item has been scheduled for forward unusability propagation (see section 4.4).

**backward_unusable** – Set when the item has been scheduled for backward unusability propagation (see section 4.4). When set, the item need not be analysed because either it is passed to a function which somehow renders the object unusable or its own function does so.

**def_checked** – This flag denotes the definition of the SSA name associated with the item has already been checked. It is unused in items which do not represent pointers. This is the only meaningful flag when the `target` of the item is set.

**malloced** – Signals that the associated object was dynamically allocated.

**param** – The items with this flag corresponds to a formal parameter of a function.

**modified** – Meaningful only in leaf items. These items have been modified by some of the functions it has been passed to.

**mod_queued** – Indicator of whether the top level item is in the backward modification propagation queue.

#### 4.1.2 Data associated with functions

Each function of the compiled program that the pass analyzes has an instance of `struct node_info` (see figure 4.3) associated with it. The name is derived from the fact that functions make up nodes in the call graph. All these structures reside in a single allocated array and the one representing a given function is selected according to the index of the corresponding call graph node. We have considered storing a pointer to this structure into the `aux` field of `struct cgraph_node` but this field is used by topological sort and so we had to avoid it.

The structure itself contains primarily pointers to items describing aggregates (see section 4.1.1). Specifically, `ptrs` points to an array of items representing all SSA names of local variables pointing to aggregates, and `loc_aggs` to an array of items describing

```
struct node_info
{
  /* The node itself. */
  struct cgraph_node *node;

  /* Flags, see comments on struct node_info_flags. */
  struct node_info_flags flags;

  /* Number of top level items corresponding to SSA pointers to aggregates. */
  int ptr_count;
  /* Dynamically allocated array of top level items corresponding to SSA
     pointers of aggregates.  */
  struct ocp_item *ptrs;

  /* Number of top level items corresponding to local aggregates. */
  int loc_agg_count;
  /* Dynamically allocated arry of top level items describing local aggregate
     variables. */
  struct ocp_item *loc_aggs;

  /* Number of parameters of this function: */
  int param_count;
  /* An array of pointers to items corresponding to formal parameters. Some
     pointers may be NULL. */
  struct ocp_item **params;

  /* Maximum value of index in items +1 */
  int max_index;
  /* Pointer to the first array item in the linked list of usable array items
     of this function. */
  struct ocp_item *arrays;

  /* Next info in the workqueue in the interprocedural stage */
  struct node_info *next;
};
```

Figure 4.3: The data structure associated with each function (call graph node).

local aggregates (i.e. not pointers). The numbers of elements of these fields are stored in `ptr_count` and `loc_agg_count` respectively. `params` points to an array of pointers to items that are associated with formal parameters of this function in the same order in which their `PARM_DECL`s appear. The number of these parameters can be obtained from `param_count`. Finally, the node info also holds flags that describe various properties of the whole function and which are listed below:

**analyzed** – The function was considered by the analysis. Functions which do not have their bodies available or which have variable number of arguments are ignored.

**ret_func_ready** – During the intraprocedural jump and return building stage, this flag means the construction of return function of this node has already started. In later stages, or when examining callees, it means the return functions are finished and available.

**queued** – This node is already present in the inteprocedural constant propagation stage work queue.

**asm_stmt** – The body of this function contains an `ASM_EXPR`.

## 4.2 Preparation Stage

The preparation stage is fairly straightforward. It starts with `preparation_stage()` allocating the array of node infos. It then traverses all call graph nodes and initializes relevant infos by calling `prepare_node()`. This function checks whether the node can be analysed and if it can, it creates all items associated with local aggregates and SSA names pointing to them, as well as pointers to parameters. Items of inappropriate types are marked as unusable during the process.

## 4.3 Intraprocedural Usability Analysis

This stage traverses all functions in the call graph and processes each one that was deemed analyzable by the previous phase by invoking `intra_usable_analysis()`. This functions performs three tasks: It examines definitions of SSA names of pointers to aggregates, setting targets of their items as required along the way. It checks their uses too and then processes all statements in the function looking for inappropriate `ADDR_EXPR`s and function calls. Whenever it finds a condition inhibiting further analysis, it sets the `unusable` flag of the associated item.

### 4.3.1 Examining SSA names definitions

Examining definitions of SSA names under scrutiny is the task of `check_ssa_def()` which has to decide whether the definition allows the SSA name to be further analysed and assign it a target if it is a new alias of another item.

If a given SSA name has a default definition, it is considered all right since it cannot create aliasing problems. Otherwise the defining statement is fetched and inspected. First,

it must be an assignment statement. Names defined by a phi-node are refused because they are likely to be a part of an algorithm impossible to analyze. Since our goal is primarily to track the `this` pointer and other similar parameters across methods and which are never involved in a phi-node. Second, the right hand side of the statement is extracted and examined, it can be of any of the following types:

- `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 is usable only if the function is some kind of malloc. See section 4.3.4 below for details.
- `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 become the target of the newly defined name. Furthermore, the pointer on the right hand side can be a result of a malloc and it have no other uses except for this statement. In that case, the item is marked as malloced (see flags in section 4.1.1). Any other case means the item in question is marked unusable.
- `NOP_EXPR`. Type conversion is the most complex case which is handled by function `check_nop_expr()`. This function handles similar cases to the three above and attempts to locate a sub-item of the requested type at the beginning of located targets. It is also capable of detecting accesses to sub-items through offset arithmetics which is generated by the C++ frontend when calling destructors of multiple ancestors.

### 4.3.2 Scrutinizing uses of SSA names

Once definitions of SSA names have been dealt with, it is necessary to make sure their uses do not leak addresses. Because calls to functions and `ADDR_EXPR`s are handled when traversing the whole function, it is only necessary to check phi-nodes and assignments at this stage. Phi-nodes are easy, any SSA name meddling with them is immediately marked unusable for the same reasons described in section 4.3.1. When examining an assignment, the following steps must be taken:

1. The left hand side is examined to find out whether it modifies any of the members of the given item. If it does, the sub-item is marked as modified (see sections 4.5 and 4.7).
2. The left hand side is inspected to determine whether this statement is a definition of an alias of the given item or a temporary SSA name which is a part of creating of such an alias. If it is so, the statement is fine and the check of this use can be terminated.
3. Otherwise, the right hand side is decomposed and searched for any undereferenced 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.

### 4.3.3 Function scan

Unfortunately, the dataflow information available for SSA names of local pointer variables is not there to help us reason about what happens to 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 their part 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 find out which items are passed as actual arguments to them, so that usability and other information can be propagated in between items representing formal and actual parameters through call graph edges. Therefore, while scanning the function, call expressions are also thoroughly examined. First, we find out whether the called function is analyzed by this pass. If it is not, all items passed to it as parameters are marked as unusable, because calling functions in other compile units and those refused in the first phase (see function 4.2) can leak object addresses and thus nothing can be assumed about such objects from that time on. One notable exception are built-in `free` function and `delete` operator which are treated as a special case (see section 4.3.4). If the function is all right, an edge info structure is allocated and assigned to the `aux` field of the `struct cgraph_edge`. Moreover, `args` field of the info is also allocated and filled with pointers to items representing actual parameters.

As a last precaution performed in this stage, the pass marks as unusable all items associated with formal parameters which are called by a function which is not analyzable or which does not represent the corresponding actual argument with an item.

```
struct edge_info
{
  /* Array of pointers to items that have been used as arguments in the call
     corresponding to this edge. Some pointers may be NULL. The size of this
     array is the number of argumnts of the call (which must be the same as
     the number of formal parameters of the callee). */
  struct ocp_item **args;

  /* Size of the following array. */
  int lat_count;

  /* Array of lattices that describe the values passed down by this edge in
     pointers to aggregates. There is one lattice per each usable item in each
     aggregate pointer parameter. The lattices describing the second aggregate
     argument follow those of the first and so on.  Within one argument, the
     lattices correspond to usable sub-items in the DFS order.  */
  struct lattice_item *lats;
};
```

Figure 4.4: The structure associated with a call graph edge.

### 4.3.4 Identifying malloc, free, new, and delete

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()`.

GCC already has a flag `ECF_MALLOC` which hints that a called function returns a pointer with no other alias, which is exactly the property we are after. Thus, checking for `malloc` is equal to finding out whether this flag of a call site is set. Unfortunately, there is no similar general-purpose flag for functions which end the lifespan of its parameter like free does. Therefore, we resort to identifying the built-in `free()` by its declaration code.

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. We have therefore decided to altered their definitions by marking them with our new special purpose function attributes `agg_safe_new` and `agg_safe_delete`. Thus, in order to determine whether a given function is either the standard `new` operator or the standard `delete` operator, we simply query the attributes for the presence of the one we are interested in.

## 4.4 Interprocedural usability analysis

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 if anything bad happens to them, 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.

Consider the example of a call graph shown in figure 4.5. 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, functions $c$ and $e$ must be prevented from optimizing because function $b$ might have given them an object after if 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.

Furthermore, there are functions, in particular $f$, $g$, $h$, $c$ 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).

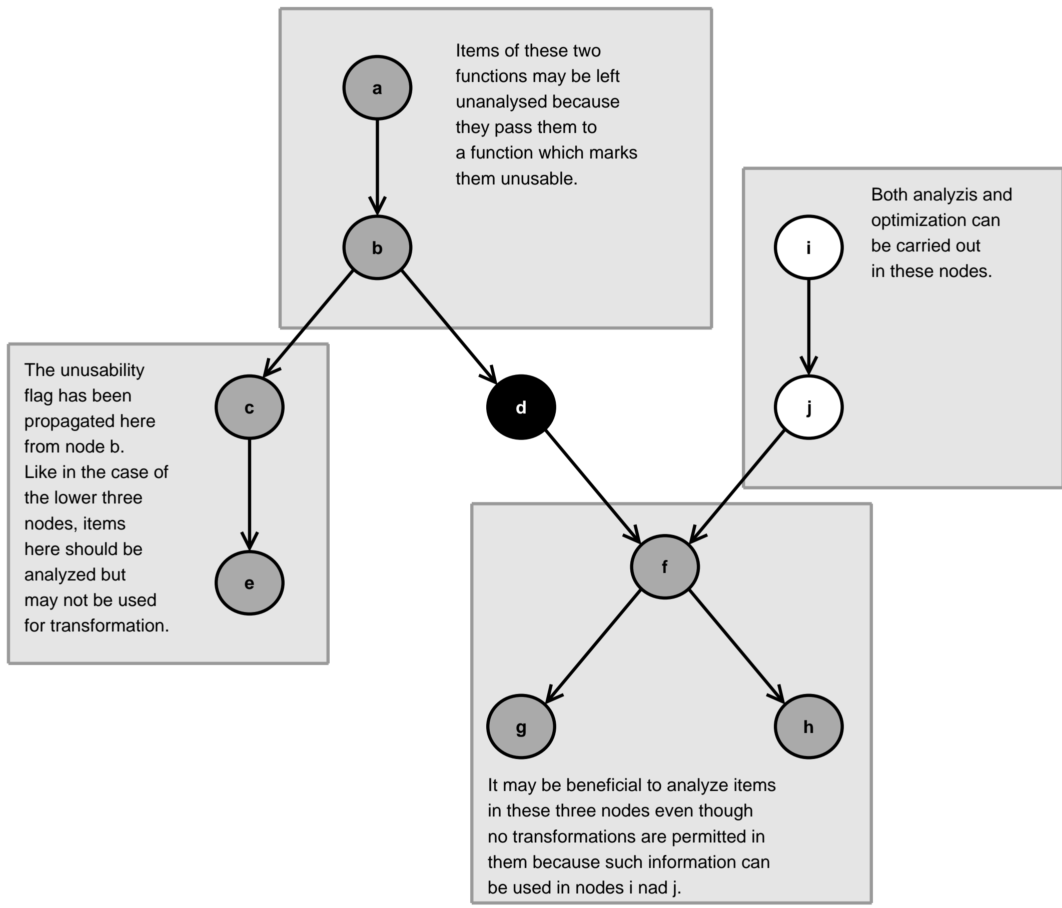



Figure 4.5: Propagation of unusability flag in a call graph. Assume each function has one parameter and passes the object it gets from callers to all of its callees and function $d$ marks the associated item unusable.

In order to propagate appropriate information to all necessary places, the interprocedural stage performs a so called *forward unusability propagation* and *backward unusability propagation*. The former is for propagating the `unusable` flag from callers to callees and the latter vice versa. Both propagations are based on work-queues containing items with the unusable flag set and which may need to be propagated. Please recall from section 4.1.1 that items have `forward_unusable` and `backward_unusable` flags which are set once the item is put into the appropriate queue and never cleared. No item needs to be added to the same queue for a second time and this flag is used by the pass to avoid it. The propagation algorithm first inserts all items found to be unusable so far to both these queues. Second, it performs the following two steps until both queues are empty:

1. The analysis processes all items from the forward propagation queue in the following way: All call sites within a function this item belongs to are examined and if the item or a part of it is passed to other functions, the items of the relevant formal parameters are also marked unusable and inserted into the forward propagation queue. Thus, this mechanism propagates unusability from callers to callees.

2. All items from the backward propagation queue are dealt with in a slightly different manner: If the item does not belong to a formal parameter, nothing needs to be done. Otherwise, all call sites in other functions that invoke the function the item belongs to are examined and the corresponding actual parameter is marked unusable and inserted into *both* forward and backward propagation queues. This rule ensures not only that unreliability information is transitively propagated to the callers but also to other callees of these callers. This is necessary because an item that might have become unreliable in one callee could afterwards be passed to another (see node $c$ in figure 4.5.

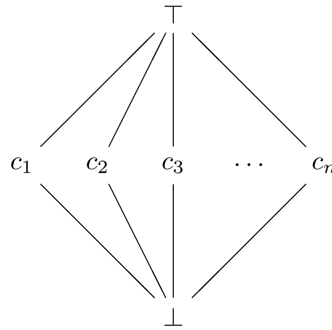
## 4.5 Interprocedural Modification Flag Propagation

The `modified` flag present in items (see section 4.1.1) also needs to be interprocedurally propagated to the callers. The mechanism is similar to the backward propagation described above except that this propagation is strictly unidirectional, this flag is never propagated forward.

## 4.6 Lattices and Their Internal Representation

The interprocedural analysis that follows this stage works with the well-known lattices for constant propagation formally defined in [37] or [30] (see figure 4.6 for a quick reminder). Internally, they are represented by instances of `struct lattice_item` given in figure 4.7. As you can see, these lattices are capable of holding a scalar constant but also a pointer to structure `array_lattice_item` which represents up to sixteen constant values stored at known constant indices in an array. The meet operations of array lattice elements with either top or bottom are the same, i.e. they are defined as the array lattice element itself and bottom respectively. Meet of two array lattice elements

is performed element-wise and known bottoms at specific indices may arise from that operation. These mini-bottoms are also created when one array lattice element has a information about an index that is not present in the other. Knowing about such index-specific bottoms is necessary for reasonable and efficient construction of return functions (see section 4.7). Last but not least, when the number of stored index related pieces of information is about to exceed sixteen, the whole lattice element is turned into a bottom. The main goal of array elements is to keep information about array sizes and strides in different dimensions. Since the number of dimensions is smaller than sixteen in all arrays in all applications we have examined, such mechanism seems feasible. Naturally, the maximum size of an array lattice element can be increased at any time.




$$\begin{aligned}
\forall c \in L : c \sqcap \top &= c \\
\forall c \in L : c \sqcap \bot &= \bot \\
\forall c \in L : c \sqcap c &= c \\
\forall c_1, c_2 \in L : c_1 \sqcap c_2 &= \bot
\end{aligned}$$

Figure 4.6: Lattices for interprocedural constant propagation.

## 4.7 Jump and Return Function Building

Once the usability and constantness properties of items have been determined, the interprocedural constant propagation of aggregate members can take place. Very much like traditional interprocedural constant propagation [12], ours starts by building jump functions. Our jump functions are also *pass through* functions as described in [12], recommended by [20] as the most cost-effective, and used by the scalar interprocedural constant propagation pass that is nowadays part of GCC. 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.)

```c
/* Type of value a lattice item stores. */
enum ocp_valtype
  {
    TOP = 0,            /* yet undetermined */
    CONST_VAL,          /* known to be const */
    CONST_ARR,          /* corresponds to array with some known vals */
    BOTTOM              /* variable or otherwise weird */
  };


/* Because a lattice never describes a single scalar constant and an array
   with known constants or bottoms at once, these are stored in the
   following union. */
union values
{
  /* The scalar constant if lattice type is CONST_VAL */
  tree const_val;

  /* Pointer to a structure describing known stuff in an array, meaningful
     if the lattice valtype is CONST_ARR.  */
  struct array_lattice_item *arr_values;
};

struct lattice_item
{
  /* contantness state of the item: */
  enum ocp_valtype valtype;
  /* See description of "union values." Is undefined if the lattice type
     is TOP or BOTTOM. */
  union values val;
};
```

Figure 4.7: Definitions for internal representation of lattices.

Nevertheless, 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, when only an item at a particular index is redefined, it may be represented by a bottom associated with that index in an array lattice item – see section 4.6.),
- was in all cases redefined with a known constant (represented by the constant) or
- was left intact by all possible control paths of the callee (represented by top).

### 4.7.1 Dataflow from the global perspective

Unlike the scalar pass, we are practically unable to build jump and return functions solely from the dataflow stored with the SSA (specifically, definitions of the names). This is impossible because we need to track the contents of individual aggregate members rather than of the pointers which reference the objects, and also the accesses to local aggregates which are not accessed through a pointer in this function.

Therefore, we perform a simple iterative data flow analysis [30] instead. Like at many other places in this pass, this data flow analysis is driven by a work list, in this particular case it is a work list of basic blocks. First, we allocate an array of lattice elements so that there is one per every usable leaf item and every basic block (`lattice_table`). 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 work list is set up so that it contains the entry block. In each step, we pop a basic block from the list and scan all its statements, updating the lattice elements as necessary (see section 4.7.2). When we are done with the statements, the control flow edges leading from the basic blocks are used to update the lattices of succeeding basic blocks. If the updates cause any change or we have hit the basic block for the first time, these basic blocks are pushed onto the work list so that they will be (re)processed later.

The traditional way of combining lattices from various control paths is the lattice meet operation. However, since we are also building the return functions, paths which do not define a particular item and those which do cannot be together represented as the constant, because the behaviour of such a combination is in fact variable. Therefore, when combining lattices during the intraprocedural analysis, any two different elements are turned into a bottom. To demonstrate this, look at figure 4.8. If the function takes the left path, one of sub-items of $a$ will be redefined, if it takes the right path, it will be not. Such behaviour must be represented by a bottom and that is what the $x$'s lattice is set to when the two paths meet. When the data flow analysis is done[3], the return function is available as the initial lattice for the exit basic block. In fact, that is why all updates to exit's lattices are written to the `retf` field of individual items rather than to the table straight away.

[3]The traditional argument why it must terminate stands, each lattice can change its value only twice and there is a finite number of them.

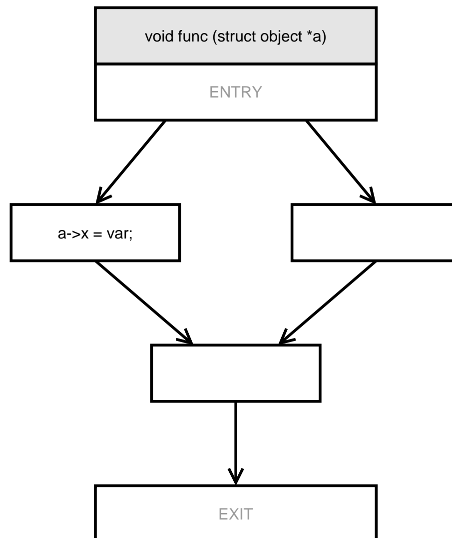



Figure 4.8: Reasons for not using traditional meet in intraprocedural DFA.

### 4.7.2 Processing individual statements

When examining what effect a statement has on the lattices discussed above, two types of statements must be considered: assignments and calls. Naturally, assignments are the most obvious way of altering a value of a part of an object. Calls are capable of doing so because objects are passed by reference and thus all modifications callees do or might do must be taken into account. Because `lattice_table` holds the initial values of lattices for each basic block, those corresponding to the analyzed one are copied to a working copy accessible through `curlats`. The values from these copies are combined with current initial lattice values of succeeding basic blocks.

Processing assignment statements is straightforward. The `analyze_modify()` function checks whether there is a usable item representing the left hand side and, if there is, determines whether the right hand side is a constant value or not. In the former case, the lattice associated with the item is assigned the constant value, in the latter, it is turned into a bottom. At the moment, the pass propagates simple integer and floating point constants and pointers to global variables and functions (what is and what is not a constant is decided by predicate `is_simple_const()`).

Dealing with call expressions is slightly more complicated. When a call is encountered, its jump function must be updated and the return function of the callee formal parameters must be superimposed on the values of lattices corresponding to items involved in the call. Obviously, that requires the called function is analyzed before this one. Since the call graph nodes are processed in topological order, this is usually the case, except for recursive functions. Whether a function has been processed and thus has its return function ready is determined by examining its `ret_func_ready` flag (see section 4.1.2). Some time ago, when the return function of a callee was not ready, we changed all lattices corresponding to the items that were passed to the function to bottom because attempting interprocedural dataflow in these cases seemed too complex and costly and the estimated benefit was low.

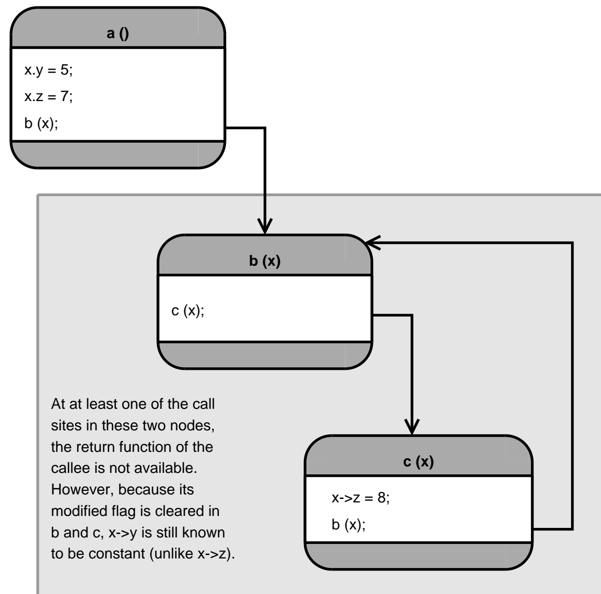



Figure 4.9: Return functions, `modified` flag and recursion.

Because this was in many cases needlessly too harsh, we have added the `modified` flag (see section 4.1.1) to each item and implemented its propagation (see section 4.5) to callers. With this mechanism, it is easy to prove that even recursive callees do not modify an item, directly or otherwise. Therefore, if there is not a return function available at some call site, only those items that have their modified flags set must be set to bottom (see figure 4.9). This provision alone has helped to increase the number of propagated scalar constants from 3200 to 3382 in FreePOOMA test suite and from 629 to 665 when running the gcc test suite.

Finally, any statement which can throw an exception that might terminate the execution of the current function must be considered a potential edge to the exit block so that the current values of lattices are involved in the construction of the return function. Lattice values both before and after the statement is processed are used so that no matter whether the exception arises before or after the statement is executed, both cases are reflected in the return function construction.

## 4.8 Interprocedural Lattice Propagation Stage

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 [12] and it closely resembles the intraprocedural constant propagation proposed by Wegman-Zadeck [37]. The algorithm keeps a work list of call graph nodes which is a FIFO structure and which is initialised by pushing all nodes on top of it in topological order. The graph entry nodes are thus processed first which leads to faster subsequent propagation. When an item is popped from the list, the current values of lattices of leaf 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.

## 4.9 Replacement Stage

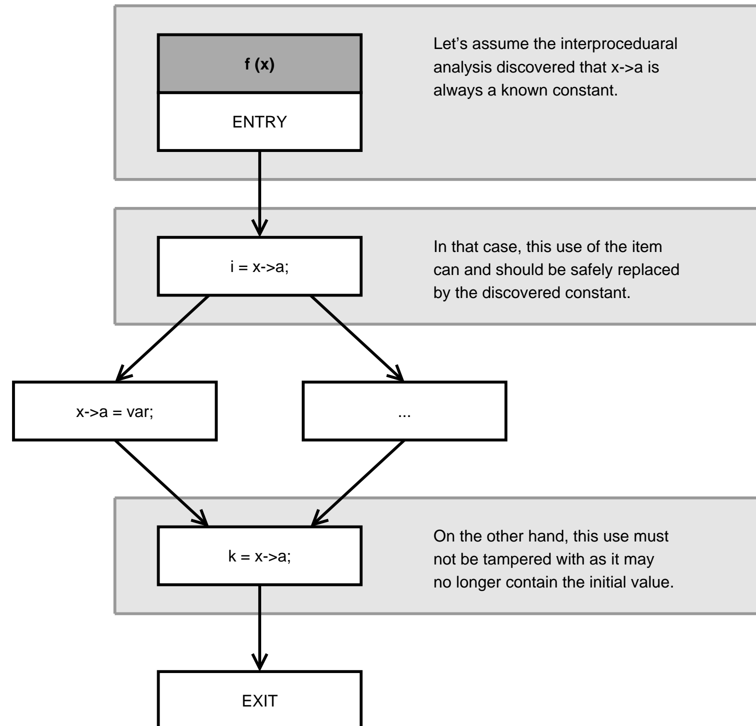



Figure 4.10: The need for data flow analysis in the replacement stage.

If some items corresponding to (a part of) a formal parameter of a function are discovered to be constant, the pass attempts to safely replace their uses in the function with the constant. The pass accomplishes it in the following steps:

1. The function is cloned and all subsequent operations are performed on the new copy. The details concerning cloning are discussed in section 4.9.1.
2. It allocates structures required for dataflow analysis like it did in the jump function building stage (see section 4.7) except the lattices corresponding to the parameters in the entry block are initialized to the values obtained by the interprocedural lattice propagation phase.
3. The same dataflow analysis is then performed, the only difference is that this time there is no need to build either the return or jump functions. The sole purpose is to obtain the state of lattices at the beginning of each basic block.
4. The pass then processes all basic blocks again. It examines the individual statements, updates lattices and when an item which is constant according to the lattices is used, it is replaced by the constant.

This mechanism guarantees that the constants will not be substituted at places where the relevant item might actually be redefined and contain a value different from the initial one (see figure 4.10).

### 4.9.1 Cloning and externally visible functions

It remains to be shown why and how a new copy of the function is obtained and that the resultant code is correct even in presence of calls from outside of this compilation unit. Let us discuss the technicalities of the cloning first. Cloning is the task of the function `clone_function()` which also has to create a new `node_info` structure on which the dataflow discussed in the previous section can operate. Along with the node information, all items must be reassociated with new declarations and new call graph edges from the cloned node to its callees must be equipped with an `edge_info` structure. The new copy of a function is created by calling `cgraph_function_versioning()` which has been modified to invoke callback functions to update the pointers to declarations of items and to deal with the new edges. Notice that there is no need to make copies of either the items or edge info structures because neither is needed at the old nodes and edges. Therefore they are not copied but simply hijacked and associated with the new entities. The only information about the old edges that must be preserved is whether there were any top lattices stored on it (see below). In that case, its `aux` pointer is set to the address of a dummy variable `may_be_redirected_ei` which is enough to encode this information.

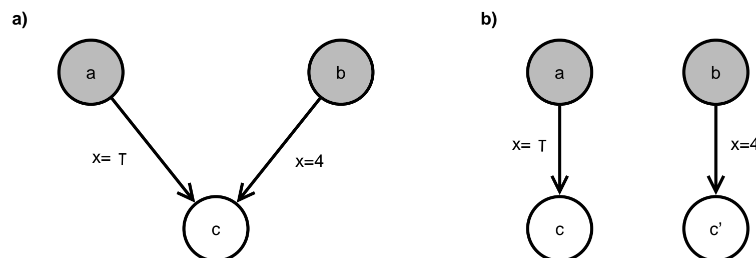



Figure 4.11: Invoking an altered method within an invalid context. The grey nodes are visible outside of this compilation unit. White ones may be but need not. **a)** Situation before replacing constants, **b)** after callgraph is updated.

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 4.11 a). When $c$ is called from $b$, $x$ is always constant. When it is called from $a$, $x$ is passed the value $a$ received from outside of this module, which is represented by a top. Therefore, the interprocedural constant propagation will decide, that when $c$ is called, $x$ is always constant since a constant met with a top is the constant. Moreover, there are less obvious situations in which similar problems arise. Have a look at the example in figure 4.12. Node $b$ can be called either from $a$ with $x$ equal to a

constant or from outside of this module with any possible value. $a$ passes $x$ to $c$ and the interprocedural lattice analysis propagates the constant there too. When uses of $x$ are substituted in a clone of $a$, that clone is inaccessible from outside of the module and thus safe. However, the original node $a$ obviously must not call the modified version of $c$ which assumes $x$ is always equal to four but the original version too. Therefore, the last step in this stage is *call graph cleanup*. 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 passe 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.

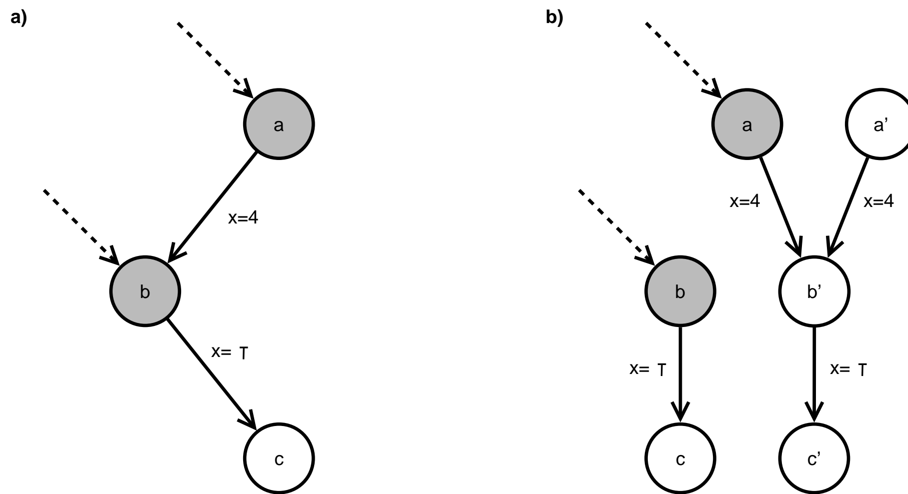



Figure 4.12: Call graph modifications. **a)** The situation before substitution, **b)** after cloning and call graph modification.

# Chapter 5

# Results

## 5.1 Correctness

Naturally, every effort was made to eliminate all possible errors or potential miscompilations. We have tested the pass thoroughly, primarily using test suits from gcc, FreePOOMA and Blitz++. As far as the gcc check is concerned, the current development tree has problems of its own. Nevertheless, even after bootstrap[1], our pass does not introduce any new errors. The FreePOOMA and Blitz++ tests or any other benchmark did not reveal any problems either.

## 5.2 Interprocedural Analysis of Aggregates and Expression Templates

In order to assess the extent to which the analysis is capable of extracting information about objects created or manipulated by expression templates, we will inspect what the pass described in the previous chapter does with the example in appendix B by looking at compiler-generated dumps. The compilation parameters were:

```
g++ -O2 -fipa-cp -static –dump-ipa-all –dump-tree-all-all array.cpp
```

Scalar constant propagation is very important because the arrays are initialised by constructors receiving such constants. The dump of the key fragment of the `main()` function is given in figure 5.1. It essentially consists of four main parts. First, a cloned version of `MyArray` is called four times to instantiate `A`, `B`, `C` and `X`. The constructor is cloned by scalar interprocedural propagation which performs the cloning for the same reasons our pass does. The following three calls to `operator=` initialize the three specified arrays. The three lines after that are the first apparent result of expression templates, they aim to create object `D.30601` which represents the calculated expression. `T.113` is a cloned version of plus operator for addition of an array and a constant. This clone is also created by scalar constant propagation. The last call to `T.115` in fact executes a new version of assignment operator which assigns the expression to the array. This function is cloned by our pass since it discovers and propagates constants in it.

[1]We carried out this test on subversion revision 123774 configured for C, C++ and Fortran.

```
                     T.114 (&A, 6, 8);
                     T.114 (&B, 6, 8);
                     T.114 (&C, 6, 8);
                     T.114 (&X, 6, 8);

                     operator= (&A, 3.0e+0);
                     operator= (&B, 4.0e+0);
                     operator= (&C, 2.0e+0);

struct MyArrayExpr   D.30488 = operator+ (&B, &C);
struct MyArrayExpr   D.30429 = T.113 (&A, 1.0e+0);
struct MyArrayExpr   D.30601 = operator* (&D.30429, &D.30488);

                     T.115 (&X, &D.30601);
```

Figure 5.1: Dump of a fragment of `main()` function.

The first question is whether all important items, in this case those representing arrays, were available in all important functions, in this case the last assignment operator (`T.115`) which contains the main loop. Unfortunately, it is immediately clear that is not the case because the items which are operands of the expression are not passed directly to the function but are encapsulated in the expression object and the analysis cannot "unpack" it.

The second important question is whether and which items have been discarded by the pass as unusable because they are used inappropriately. `main()` does not contain any such statement and so we can go on to examine the other functions. The constructor and scalar assignments are both fine, because they do not manipulate with the addresses of the objects at their disposal. On the other hand, the two plus operators do cause problems in this respect. The address of the array is stored in a field of the temporary `LiteralExpr` object (see figure 5.2). Such undereferenced use makes the analysis mark the corresponding item as unusable because it assumes the address might have escaped its control (see section 4.3.2). Interprocedural propagation of the unusability flag will then make the items corresponding to the three arrays unusable at all places in the program. This is not actually a problem in this simple example but it may be in more complex cases.

```
struct LiteralExpr   D.31558.val = 1.0e+0;
struct IdExpr        D.31557.arr = x_1(D);
struct MyArray &     SR.108_2 = D.31557.arr;
                     bin.left.arr = SR.108_2;
double               SR.109_3 = D.31558.val;
                     bin.right.val = SR.109_3;
                     r.op = bin;
struct MyArrayExpr   D.31556 = r;
                     return D.31556;
```

Figure 5.2: Abridged dump of `operator+` function for an array and a scalar constant.

On the contrary, the remaining array in this example, `X` is both available and perfectly usable in the assignment operator. Moreover, the constant propagation is able to

determine that strides and sizes are constant and replaces their uses with the determined values. Therefore, we can conclude that even though we have successfully managed to analyze and extract information from within objects that are passed around in between different functions as the `this` pointer (see discussion at the beginning of chapter 4), that is not enough to fully optimize code generated by expression templates. Nevertheless, the existing code can certainly serve as the basis for future development to overcome these difficulties. In particular, the following issues will have to be addressed:

1. Plus operators store addresses of arrays into a field within a local aggregate. This must be allowed if it can be proved the address does not escape uncontrollably.
2. The function then copies that address in between members of different aggregate variables (see figure 5.2). Either the pass must be adapted so that it can follow the flow of these pointers or another, simpler pass that will forward propagate such addresses must be run earlier. Given the structures have no aliases outside of this function, it should definitely be feasible.
3. All operators return their expression objects by value using the return statement. However, the current version of the analysis cannot handle the return statement, let alone returns by value. Support of both must be added, probably in the form of yet another kind of return functions that will describe the resultant object in terms of obtained parameters and constants. Moreover, means to propagate unusability even through these new return functions in both directions must be established.
4. `main()` function passes both results of addition to the multiplication operator for re-encapsulation. It is necessary to propagate the arrays to the function too as some virtual arguments. The same applies to the call of the assignment operator.
5. Whether a leaf item contains an address of another item is context sensitive information and must be treated that way.

Given these requirements, probably the most appealing proposal is to run significant part of the current pass twice. The iteration will primarily need to assess whether leaf-items representing pointers can potentially safely have targets while the second run will actually set up virtual arguments and proceed with true constant propagation.

## 5.3 Benchmarks

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 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.

The number of replacements that have been performed during various compilations of different code are given in table 5.1. 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 [2] test suite | 3367 | 12 | 3379 |
| Tramp3D [21] | 2 | 0 | 2 |
| DLV [27] | 118 | 0 | 118 |
| Blitz++ [33] test suite | 96 | 0 | 96 |

Table 5.1: Number of replacements performed.

In order to measure the impact on execution time, we have run several benchmarks including Tramp3D [21], DLV [27], Blitz++ [33] acoustic 3D benchmark and a slightly modified version of our example given in appendix B (the arrays are a million times bigger and the expression is performed twenty times). In all these cases, we have configured GCC to optimization level 2 with loop unrolling, scalar interprocedural constant propagation and static linking. Blitz++ acoustic 3D and our array example do not accept any command line parameters. DLV was run through a benchmarking script and Tramp3D was launched in the following way:

```
tramp3d –cartvis 1.0 0.0 –rhomin 1e-8 -n 1000
```

DLV results are presented in table 5.3, the rest of the benchmarks in table 5.2. As you can see, the performance gains are modest or none at all. On the other hand, the technique works and we believe once it is extended to provide for expression templates' right hand sides as described in the previous section, the improvement will be significant because it will expose far more opportunities for better loop optimizations. Moreover, the interprocedural propagation of constants within aggregates is in many ways more powerful and capable than its scalar interprocedural counterpart that is present in GCC today, and the interprocedural analysis of aggregates alone has great potential to be successfully used in many ways.

| Benchmark | Unpatched | Patched |
|---|---|---|
| Tramp 3D | 11m7.27s | 11m6.55s |
| Blitz++ acoustic 3D (raw) | 7.25s | 7.14s |
| Modified array example | 25.45s | 24.87s |

Table 5.2: 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 run-time of the compiler and allocates slightly over 1MB when compiling Tramp3D.

| Benchmark | Unpatched | Patched |
|---|---|---|
| stratcomp1-all | 1.59s | 1.56s |
| stratcomp-770.2-q | 0.41s | 0.39s |
| 2qbf1 | 8.71s | 8.68s |
| primeimpl2 | 5.89s | 5.78s |
| ancestor | 86.82s | 82.74s |
| 3col-simplex1 | 4.26s | 4.17s |
| 3col-ladder | 97.37s | 93.72s |
| 3col-random1 | 6.89s | 6.77s |
| hp-random1 | 8.15s | 8.08s |
| hamcycle-free | 0.86s | 0.83s |
| decomp2 | 7.07s | 6.87s |
| bw-p4-esra-a | 39.21s | 38.61s |
| bw-p5-nopush | 3.64s | 3.38s |
| bw-p5-pushbin | 3.24s | 3.10s |
| bw-p5-nopushbin | 1.01s | 1.00s |
| 3sat-1 | 18.68s | 18.89s |
| 3sat-1-constraint | 9.99s | 9.77s |
| hanoi-towers | 2.15s | 2.17s |
| ramsey | 3.93s | 3.62s |
| cristal | 6.38s | 6.09s |
| hanoi-k | 17.23s | 16.62s |
| 21-queens | 4.82s | 4.70s |
| mstdir | 7.12s | 6.99s |
| mstundir | 70.97s | 69.99s |
| timetabling | 5.00s | 4.72s |

Table 5.3: Execution times of DLV [27] benchmarks

# Chapter 6

# Conclusion

In this work we have described a number of aspects concerning C++ scientific code, especially when it is compiled by GCC compiler. After a short comparison with Fortran in chapter 1, we have demonstrated what scientific code looks like in chapter 2. In particular, we have shown how template expressions are used to eliminate creation of temporary variables which can grossly prevent efficient calculations. We have also briefly discussed some non-standard features for GCC intended for scientific software and introduced two representative libraries for the same purpose.

We have then moved on to look under the hood of the compiler to examine what techniques are nowadays used to counter the issues arising from expression templates and removing piling layers of abstraction in general. We have primarily considered inlining because it is the basic mechanism of removing abstraction, especially when the compiled program consists of a great number of tiny functions, which is the typical case with expression templates. We have used the example of Tramp3D compilation to show the importance and power of inlining as well as its limits. We briefly discussed the greedy algorithm its implementation is based on and described the concept and advantages of early inlining.

We have then investigated two interprocedural alternatives to inlining: the interprocedural constant propagation and function cloning. We have briefly outlined the typical algorithm to achieve the former because many of the concepts were used in the pass described in chapter four. Because function cloning is not implemented in GCC, we have discussed its principles and caveats described in literature. At the intraprocedural level, we dealt with static replacement of aggregates first. We have briefly explained how breaking up of aggregates is performed in GCC and demonstrated that more aggressive inlining exposes opportunities for more scalar replacements but at the same time the number of aggregates which cannot be scalarized because of aliasing also increases. Finally, we discussed the core of compilation of scientific code, the loop transformations. Apart from giving a number of examples of loop optimizations already present in GCC, we have stressed the benefits of propagating constants to dependence analysis, induction variable optimization and others.

Chapter 4 is the most significant one. It describes the implementation of *interprocedural analysis of aggregates* we have proposed and implemented in GCC 4.3. The aim of this analysis is to identify well-behaving objects such as those that are only passed around

as `this` pointers, track how they are passed in between different functions and allow safe extraction of information. In order to demonstrate the usefulness and correctness of the analysis, we have used it to implement interprocedural propagation of constants within aggregates. The chapter describes the core data structures, and all stages of the pass in detail, explaining the main mechanism involved and issues tackled.

Chapter 5 discusses how well this technique deals with code resulting from extensive use of expression templates. In a simple case study, we have shown the pass works as expected and does propagate constants within aggregates interprocedurally. On the other hand, we have also presented yet unsolved problems arising from the existence of complex object representing expressions. Perhaps more importantly, we identified the elementary obstacles and proposed a way forward. Furthermore, behaviour of the pass is demonstrated on a number of benchmarks, even though the noticable improvements are modest. Despite that we do believe the current state of the analysis is a solid basis for future development, whether related to constant propagation or not.

# Appendix A

# Naive Implementation of Array Expressions in C++

This appendix contains full source code of "naive array expressions implementation" discussed in section 2.1.1. It is a very class and a number of operators implementing addition and multiplication in the most simple way.

```
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include <iostream>

#define R1 6
#define R2 7

using namespace std;

class TradArray
{
private:
  // pointer to allocated memory which contains data
  double *data;
  // sizes in the two dimensions
  int dims[2];
  // number of allocated doubles
  int size;

  // The default constructor is private so that other constructors must be
  // used to instantiate the object.
  TradArray ()
  {}

public:
  // The standard constructor accepting array dimensions as parameters
  TradArray (int r0, int r1);

  // Copy constructor
  TradArray (const TradArray &A);
```

```
  // Destructor
  ~TradArray ();

  int getDimension(int d)
  {
    return dims[d];
  }

  // Individual array elements are accesses in function (or Fortran) like
  // fashion.
  double &operator()(int i0, int i1)
  {
    return data[i0 * dims[1] + i1];
  }

  // The constant variant of the above operator
  const double &operator()(int i0, int i1) const
  {
    return data[i0 * dims[1] + i1];
  }

  // Accessor to the dimensions array.
  int getDim(int i) const
  {
    return dims[i];
  }

  // Assignment operator  that copies the data  in array a  to data of
  // this object. Both this objects must have the same dimensions.
  TradArray &operator=(const TradArray &a);

  // Assignment operator that fills the entire array with the constant x.
  TradArray &operator=(const double x);
};

// Operator for adding elements of two arrays. Both arrays must have the same
// dimensions.  The result is a new temporary array containing the result.
TradArray operator+(const TradArray &l, const TradArray &r);

// The following two addition operators serve to add constants to all elements
// of an array. Again, the result is a temporary object.
TradArray operator+(const double l, const TradArray &r);
TradArray operator+(const TradArray &l, const double r);

// The same set od operators for multiplication.
TradArray operator*(const TradArray &l, const TradArray &r);
TradArray operator*(const double l, const TradArray &r);
TradArray operator*(const TradArray &l, const double r);

// Similar operator overrides could be provided for any other binary
// operations one would like to implement on arrays.

// Function implementing a unary square root operation on all elements of the
// given array. It also returns a temporary object.
TradArray sqrt(const TradArray &x);
```

```
TradArray::TradArray (int r0, int r1)
{
  cout << "Ordinary␣constructor␣called" << endl;

  dims[0] = r0;
  dims[1] = r1;
  size = r0 * r1;
  data = new double[size];
}

TradArray::TradArray (const TradArray &a)
{
  cout << "Copy␣constructor␣called" << endl;

  dims[0] = a.dims[0];
  dims[1] = a.dims[1];
  size = a.size;

  data = new double[size];
  memcpy (a.data, data, size * sizeof (double));
}

TradArray::~TradArray ()
{
  cout << "Destructor␣called" << endl;

  delete[] data;
}

TradArray &TradArray::operator=(const TradArray &a)
{
  memcpy (data, a.data, size * sizeof (double));
  return *this;
}

TradArray &TradArray::operator=(const double x)
{
  int i, j;

  for (i = 0; i < dims[0]; i++)
    for (j = 0; j < dims[1]; j++)
      data[i * dims[1] + j] = x;

  return *this;
}

TradArray operator+(const TradArray &l, const TradArray &r)
{
  int d0 = l.getDim(0);
  int d1 = l.getDim(1);
  TradArray a(d0, d1);
  int i, j;

  for (i = 0; i < d0; i++)
    for (j = 0; j < d1; j++)
      a(i,j) = l(i,j) + r(i,j);
```

```
  return a;
}

TradArray operator+(const double l, const TradArray &r)
{
  int d0 = r.getDim(0);
  int d1 = r.getDim(1);
  TradArray a(d0, d1);
  int i, j;

  for (i = 0; i < d0; i++)
    for (j = 0; j < d1; j++)
      a(i,j) = l + r(i,j);

  return a;
}

TradArray operator+(const TradArray &l, const double r)
{
  int d0 = l.getDim(0);
  int d1 = l.getDim(1);
  TradArray a(d0, d1);
  int i, j;

  for (i = 0; i < d0; i++)
    for (j = 0; j < d1; j++)
      a(i,j) = l(i,j) + r;

  return a;
}

TradArray operator*(const TradArray &l, const TradArray &r)
{
  int d0 = l.getDim(0);
  int d1 = l.getDim(1);
  TradArray a(d0, d1);
  int i, j;

  for (i = 0; i < d0; i++)
    for (j = 0; j < d1; j++)
      a(i,j) = l(i,j) * r(i,j);
  return a;
}

TradArray operator*(const double l, const TradArray &r)
{
  int d0 = r.getDim(0);
  int d1 = r.getDim(1);
  TradArray a(d0, d1);
  int i, j;

  for (i = 0; i < d0; i++)
    for (j = 0; j < d1; j++)
      a(i,j) = l * r(i,j);

  return a;
}
```

```
TradArray operator*(const TradArray &l, const double r)
{
  int d0 = l.getDim(0);
  int d1 = l.getDim(1);
  TradArray a(d0, d1);
  int i, j;

  for (i = 0; i < d0; i++)
    for (j = 0; j < d1; j++)
      a(i,j) = l(i,j) * r;

  return a;
}

TradArray sqrt(const TradArray &x)
{
  int d0 = x.getDim(0);
  int d1 = x.getDim(1);
  TradArray a(d0, d1);
  int i, j;

  for (i = 0; i < d0; i++)
    for (j = 0; j < d1; j++)
      a(i,j) = sqrt(x(i,j));

  return a;
}

int main (int argc, char *argv[])
{
  TradArray A(R1, R2), B(R1, R2), C(R1, R2);
  TradArray X(R1, R2);
  int i, j;

  A = 3;
  B = 4;
  C = 2;

  cout << "Starting␣the␣evaluation..." << endl;

  X = (A + 1) * (B + C);

  cout << "Expression␣evaluation␣finished" <<  endl << endl;

  for (i = 0; i < R1; i++)
    {
      for (j = 0; j < R2; j++)
        {
          cout << X(i, j) << "␣␣";
        }
      cout << endl;
    }

  return 0;
}
```

# Appendix B

# Using Expression Templates to Implement Array Expressions

This appendix demonstrates how expression templates are used to implement array expressions without the need for costly temporary objects. Implementation details are discussed in section 2.1.2.

```
#include <stdlib.h>
#include <math.h>
#include <assert.h>

#include <iostream>

#define R1 6
#define R2 7
#define R3 8

using namespace std;

template<typename Operation>
class MyArrayExpr;

// N can be in the range of 1 to 3
template<int N>
class MyArray {
private:
  // pointer to allocated memory which holds data
  double *data;
  // size of the allocated data in doubles
  int size;
  // sizes of the individual dimensions
  int dims[N];
  // dimension access strides
  int strides[N];

  // The default constructor is private so that other constructors must be
  // used to instantiate the object.
  MyArray ()
  {}
```

```
public:
  // Constructors allocating memory, one for each supported rank
  MyArray (int r0);
  MyArray (int r0, int r1);
  MyArray (int r0, int r1, int r2);

  // Copy constructor
  MyArray (const MyArray &a);

  // Destructor
  ~MyArray ();

  int getDimension(int d)
  {
    return dims[d];
  }

  // Data access method. Two for each rank, one constant and one that is not
  double &operator()(int i0)
  {
    assert (N == 1);
    return data[i0];
  }

  double &operator()(int i0, int i1)
  {
    assert (N == 2);
    return data[i0 * strides[0] + i1 * strides[1]];
  }

  double &operator()(int i0, int i1, int i2)
  {
    assert (N == 3);
    return data[i0 * strides[0] + i1 * strides[1] + i2 * strides[2]];
  }

  const double &operator()(int i0) const
  {
    assert (N == 1);
    return data[i0];
  }

  const double &operator()(int i0, int i1) const
  {
    assert (N == 2);
    return data[i0 * strides[0] + i1 * strides[1]];
  }

  const double &operator()(int i0, int i1, int i2) const
  {
    assert (N == 3);
    return data[i0 * strides[0] + i1 * strides[1] + i2 * strides[2]];
  }

  // Assignment operators
  MyArray &operator=(const MyArray<N> &a);
```

```
  MyArray &operator=(const double x);

  // This assignemnt operator accepts axpression template objects
  template<typename Operation>
  MyArray &operator= (const MyArrayExpr<Operation> &expr);
};


// Simple objects defining binary element-wise operations
class AddOp
{
public:
  static double evaluate (double x, double y)
  {
    return x + y;
  }
};

class MultiplyOp
{
public:
  static double evaluate (double x, double y)
  {
    return x * y;
  }
};

// A simple object defining unary operation
class SqrtOp
{
public:
  static double evaluate (double x)
  {
    return sqrt(x);
  }
};


// Encapsulator of unary operations
template<typename Expr, typename Operand>
class UnaryExpr
{
private:
  Operand op;

public:
  UnaryExpr (const Operand &op_) : op (op_)
  { }

  // Apply methods returns result for the given index, there is a version
  // foreach rank
  double apply (int i0) const
  {
    return Expr::evaluate (op.apply (i0));
  }
```

```
  double apply (int i0, int i1) const
  {
    return Expr::evaluate (op.apply (i0, i1));
  }

  double apply (int i0, int i1, int i2) const
  {
    return Expr::evaluate (op.apply (i0, i1, i2));
  }
};

// Encapsulator of binary operations
template<typename Expr, typename Left, typename Right>
class BinaryExpr
{
private:
  Left left;
  Right right;

public:
  BinaryExpr (const Left &left_, const Right &right_) :
    left (left_), right (right_)
  { }

  // Apply methods returns result for the given index, there is a version
  // foreach rank
  double apply (int i0) const
  {
    return Expr::evaluate (left.apply (i0), right.apply (i0));
  }

  double apply (int i0, int i1) const
  {
    return Expr::evaluate (left.apply (i0, i1), right.apply (i0, i1));
  }

  double apply (int i0, int i1, int i2) const
  {
    return Expr::evaluate (left.apply (i0, i1, i2), right.apply (i0, i1, i2));
  }
};

// Array reference holder
template<int N>
class IdExpr
{
private:
  const MyArray<N> &arr;
public:
  IdExpr (const MyArray<N> &arr_) : arr (arr_)
  { }

  // apply method simply returns the element of the array, one version per rank
  double apply (int i0) const
  {
    return arr (i0);
  }
```

```
  double apply (int i0, int i1) const
  {
    return arr (i0, i1);
  }

  double apply (int i0, int i1, int i2) const
  {
    return arr (i0, i1, i2);
  }
};

// Double scalar constant encapsulator
class LiteralExpr
{
private:
  double val;

public:
  LiteralExpr (const double val_) : val (val_)
  { }

  // apply methods return the constant, there still need to be three versions
  double apply (int i0) const
  {
    return val;
  }

  double apply (int i0, int i1) const
  {
    return val;
  }

  double apply (int i0, int i1, int i2) const
  {
    return val;
  }
};

// Auxiliary expression encapsulator
template<typename Operation>
class MyArrayExpr
{
private:
  Operation op;
public:

  MyArrayExpr (const Operation &op_) :
    op (op_)
  {  }

  double apply (int i0) const
  {
    return op.apply (i0);
  }
```

```
  double apply (int i0, int i1) const
  {
    return op.apply (i0, i1);
  }

  double apply (int i0, int i1, int i2) const
  {
    return op.apply (i0, i1, i2);
  }
};


// Operator overloads building expression objects:
template<typename A, typename B>
MyArrayExpr<BinaryExpr<AddOp, MyArrayExpr<A>, MyArrayExpr<B> > >
operator+ (const MyArrayExpr<A> &x, const MyArrayExpr<B> &y)
{
  BinaryExpr<AddOp, MyArrayExpr<A>, MyArrayExpr<B> > bin(x, y);
  MyArrayExpr<BinaryExpr<AddOp, MyArrayExpr<A>, MyArrayExpr<B> > > r(bin);
  return r;
}

template<int N, typename B>
MyArrayExpr<BinaryExpr<AddOp, IdExpr<N>, MyArrayExpr<B> > >
operator+ (const MyArray<N> &x, const MyArrayExpr<B> &y)
{
  BinaryExpr<AddOp, IdExpr<N>, MyArrayExpr<B> > bin(x, y);
  MyArrayExpr<BinaryExpr<AddOp, IdExpr<N>, MyArrayExpr<B> > > r(bin);
  return r;
}

template<typename A, int N>
MyArrayExpr<BinaryExpr<AddOp, MyArrayExpr<A>, IdExpr<N> > >
operator+ (const MyArrayExpr<A> &x, const MyArray<N> &y)
{
  BinaryExpr<AddOp, MyArrayExpr<A>, IdExpr<N> > bin(x, y);
  MyArrayExpr<BinaryExpr<AddOp, MyArrayExpr<A>, IdExpr<N> > > r(bin);
  return r;
}

template<int N>
MyArrayExpr<BinaryExpr<AddOp, IdExpr<N>, IdExpr<N> > >
operator+ (const MyArray<N> &x, const MyArray<N> &y)
{
  BinaryExpr<AddOp, IdExpr<N>, IdExpr<N> > bin(x, y);
  MyArrayExpr<BinaryExpr<AddOp, IdExpr<N>, IdExpr<N> > > r(bin);
  return r;
}

template<typename B>
MyArrayExpr<BinaryExpr<AddOp, LiteralExpr, MyArrayExpr<B> > >
operator+ (const double x, const MyArrayExpr<B> &y)
{
  BinaryExpr<AddOp, LiteralExpr, MyArrayExpr<B> > bin(x, y);
  MyArrayExpr<BinaryExpr<AddOp, LiteralExpr, MyArrayExpr<B> > > r(bin);
  return r;
}
```

```
template<typename A>
MyArrayExpr<BinaryExpr<AddOp, MyArrayExpr<A>, LiteralExpr> >
operator+ (const MyArrayExpr<A> &x, const double y)
{
  BinaryExpr<AddOp, MyArrayExpr<A>, LiteralExpr > bin(x, y);
  MyArrayExpr<BinaryExpr<AddOp, MyArrayExpr<A>, LiteralExpr > > r(bin);
  return r;
}

template<int N>
MyArrayExpr<BinaryExpr<AddOp, LiteralExpr, IdExpr<N> > >
operator+ (const double x, const MyArray<N> &y)
{
  BinaryExpr<AddOp, LiteralExpr, IdExpr<N> > bin(x, y);
  MyArrayExpr<BinaryExpr<AddOp, LiteralExpr, IdExpr<N> > > r(bin);
  return r;
}

template<int N>
MyArrayExpr<BinaryExpr<AddOp, IdExpr<N>, LiteralExpr> >
operator+ (const MyArray<N> &x, const double y)
{
  BinaryExpr<AddOp, IdExpr<N>, LiteralExpr > bin (x, y);
  MyArrayExpr<BinaryExpr<AddOp, IdExpr<N>, LiteralExpr > > r (bin);
  return r;
}

template<typename A, typename B>
MyArrayExpr<BinaryExpr<MultiplyOp, MyArrayExpr<A>, MyArrayExpr<B> > >
operator* (const MyArrayExpr<A> &x, const MyArrayExpr<B> &y)
{
  BinaryExpr<MultiplyOp, MyArrayExpr<A>, MyArrayExpr<B> > bin(x, y);
  MyArrayExpr<BinaryExpr<MultiplyOp, MyArrayExpr<A>, MyArrayExpr<B> > > r(bin);
  return r;
}

template<int N, typename B>
MyArrayExpr<BinaryExpr<MultiplyOp, IdExpr<N>, MyArrayExpr<B> > >
operator* (const MyArray<N> &x, const MyArrayExpr<B> &y)
{
  BinaryExpr<MultiplyOp, IdExpr<N>, MyArrayExpr<B> > bin(x, y);
  MyArrayExpr<BinaryExpr<MultiplyOp, IdExpr<N>, MyArrayExpr<B> > > r(bin);
  return r;
}

template<typename A, int N>
MyArrayExpr<BinaryExpr<MultiplyOp, MyArrayExpr<A>, IdExpr<N> > >
operator* (const MyArrayExpr<A> &x, const MyArray<N> &y)
{
  BinaryExpr<MultiplyOp, MyArrayExpr<A>, IdExpr<N> > bin(x, y);
  MyArrayExpr<BinaryExpr<MultiplyOp, MyArrayExpr<A>, IdExpr<N> > > r(bin);
  return r;
}
```

```
template<int N>
MyArrayExpr<BinaryExpr<MultiplyOp, IdExpr<N>, IdExpr<N> > >
operator* (const MyArray<N> &x, const MyArray<N> &y)
{
  BinaryExpr<MultiplyOp, IdExpr<N>, IdExpr<N> > bin(x, y);
  MyArrayExpr<BinaryExpr<MultiplyOp, IdExpr<N>, IdExpr<N> > > r(bin);
  return r;
}

template<typename B>
MyArrayExpr<BinaryExpr<MultiplyOp, LiteralExpr, MyArrayExpr<B> > >
operator* (const double x, const MyArrayExpr<B> &y)
{
  BinaryExpr<MultiplyOp, LiteralExpr, MyArrayExpr<B> > bin(x, y);
  MyArrayExpr<BinaryExpr<MultiplyOp, LiteralExpr, MyArrayExpr<B> > > r(bin);
  return r;
}

template<typename A>
MyArrayExpr<BinaryExpr<MultiplyOp, MyArrayExpr<A>, LiteralExpr> >
operator* (const MyArrayExpr<A> &x, const double y)
{
  BinaryExpr<MultiplyOp, MyArrayExpr<A>, LiteralExpr > bin(x, y);
  MyArrayExpr<BinaryExpr<MultiplyOp, MyArrayExpr<A>, LiteralExpr > > r(bin);
  return r;
}

template<int N>
MyArrayExpr<BinaryExpr<MultiplyOp, LiteralExpr, IdExpr<N> > >
operator* (const double x, const MyArray<N> &y)
{
  BinaryExpr<MultiplyOp, LiteralExpr, IdExpr<N> > bin(x, y);
  MyArrayExpr<BinaryExpr<MultiplyOp, LiteralExpr, IdExpr<N> > > r(bin);
  return r;
}

template<int N>
MyArrayExpr<BinaryExpr<MultiplyOp, IdExpr<N>, LiteralExpr> >
operator* (const MyArray<N> &x, const double y)
{
  BinaryExpr<MultiplyOp, IdExpr<N>, LiteralExpr > bin(x, y);
  MyArrayExpr<BinaryExpr<MultiplyOp, IdExpr<N>, LiteralExpr > > r(bin);
  return r;
}

template<typename A>
MyArrayExpr<UnaryExpr<SqrtOp, MyArrayExpr<A> > >
sqrt (const MyArrayExpr<A> &x)
{
  UnaryExpr<SqrtOp, MyArrayExpr<A> > bin (x);
  MyArrayExpr<UnaryExpr<SqrtOp, MyArrayExpr<A> > > r (bin);

  return r;
}
```

```
template<int N>
MyArrayExpr<UnaryExpr<SqrtOp, IdExpr<N> > >
sqrt (const MyArray<N> &x)
{
  UnaryExpr<SqrtOp, IdExpr<N> > bin (x);
  MyArrayExpr<UnaryExpr<SqrtOp, IdExpr<N> > > r (bin);

  return r;
}

template<int N>
MyArray<N>::MyArray(int r0)
{
  assert (N == 1);
  dims[0] = r0;
  strides[0] = 0;
  size = r0;
  data = new double[size];
}

template<int N>
MyArray<N>::MyArray (int r0, int r1)
{
  assert (N == 2);
  dims[0] = r0;
  dims[1] = r1;
  strides[0] = r1;
  strides[1] = 1;
  size = r0 * r1;
  data = new double[size];
}

template<int N>
MyArray<N>::MyArray (int r0, int r1, int r2)
{
  assert (N == 3);
  dims[0] = r0;
  dims[1] = r1;
  dims[2] = r2;
  strides[0] = r1 * r2;
  strides[1] = r2;
  strides[2] = 1;
  size = r0 * r1 * r2;
  data = new double[size];
}

template<int N>
MyArray<N>::MyArray(const MyArray<N> &a)
{
  int i;

  for (i = 0; i < N; i++)
    {
      dims[i] = a.dims[i];
      strides[i] = a.strides[i];
    }
```

```
  size = a.size;
  data = new double [size];
  memcpy (data, a.data, size * sizeof (double));
}

template<int N>
MyArray<N>::~MyArray ()
{
  delete[] data;
}

template<int N>
MyArray<N> &MyArray<N>::operator=(const MyArray<N> &a)
{
  memcpy (data, a.data, size * sizeof (double));
  return *this;
}

// This method will never be used, see the specializations below
template<int N>
MyArray<N> &MyArray<N>::operator=(const double x)
{
  assert (0);
  return *this;
}

template<>
MyArray<1> &MyArray<1>::operator=(const double x)
{
  int i;
  for (i = 0; i < dims[0]; i++)
    data[i * strides[0]] = x;

  return *this;
}

template<>
MyArray<2> &MyArray<2>::operator=(const double x)
{
  int i, j;

  for (i = 0; i < dims[0]; i++)
    for (j = 0; j < dims[1]; j++)
      data[i * strides[0] + j * strides[1]] = x;

  return *this;
}

template<>
MyArray<3> &MyArray<3>::operator=(const double x)
{
  int i, j, k;

  for (i = 0; i < dims[0]; i++)
    for (j = 0; j < dims[1]; j++)
      for (k = 0; k < dims[2]; k++)
        data[i * strides[0] + j * strides[1] + k * strides[2]] = x;
```

```
  return *this;
}

// This method will never be used, see the specializations below
template<int N>
template<typename Operation>
MyArray<N> &MyArray<N>::operator= (const MyArrayExpr<Operation> &expr)
{
  assert (0);
  return *this;
}

// Assignment operators accepting expression objects:
template<>
template<typename Operation>
MyArray<1> &MyArray<1>::operator= (const MyArrayExpr<Operation> &expr)
{
  int i;

  for (i = 0; i < dims[0]; i++)
    data[i * strides[0]] = expr.apply (i);

  return *this;
}

template<>
template<typename Operation>
MyArray<2> &MyArray<2>::operator= (const MyArrayExpr<Operation> &expr)
{
  int i, j;

  for (i = 0; i < dims[0]; i++)
    for (j = 0; j < dims[1]; j++)
      data[i * strides[0] + j * strides[1]] = expr.apply (i, j);

  return *this;
}

template<>
template<typename Operation>
MyArray<3> &MyArray<3>::operator= (const MyArrayExpr<Operation> &expr)
{
  int i, j, k;
  double v;

  for (i = 0; i < dims[0]; i++)
    for (j = 0; j < dims[1]; j++)
      for (k = 0; k < dims[2]; k++)
        {
          v = expr.apply (i, j, k);
          data[i * strides[0] + j * strides[1] + k * strides[2]] = v;
        }

  return *this;
}
```

```
// The main program calculates a simple expression, prints out the result and
// exits.
int main (int argc, char *argv[])
{
  MyArray<2> A(R1, R2), B(R1, R2), C(R1, R2);
  MyArray<2> X(R1, R2);
  int i, j;

  A = 3;
  B = 4;
  C = 2;

  X = (A + 1) * (B + C);

  cout << endl;

  for (i = 0; i < R1; i++)
    {
      for (j = 0; j < R2; j++)
        {
          cout << X(i, j) << "  ";
        }
      cout << endl;
    }

  return 0;
}
```

# Appendix C

# Demonstration of Template Expression Code Efficiency

The following is a slightly abbreviated tree dump of `operator=(const MyArray <N> &a)` function generated by gcc 4.1.2 by running g++ -O1 –dump-ipa-all –dump-tree-all array.cpp (The -O1 performs inlining of function definined within a class or those with `inline` function specifier but does no further automatic inlining). The source code used is the same as presented in appendix B except that the `operator()` methods have been modified not to be inline functions.

```
<bb 0>:
  if (this->dims[0] > 0) goto <L23>; else goto <L5>;
<L23>:;
  i = 0;
  goto <bb 3> (<L2>);
<L22>:;
  j = 0;

<L1>:;
  D.32970 = this->data + (double *) ((unsigned int) (this->strides[0] * i +
              this->strides[1] * j) * 8);
  this = &expr->op;
  this = &this->right.op;
  D.33796 = operator() (this->right.arr, i, j);
  y = *D.33796;
  D.33801 = operator() (this->left.arr, i, j);
  y = y + *D.33801;
  this = &this->left.op;
  y = this->right.val;
  D.33823 = operator() (this->left.arr, i, j);
  *D.32970 = y * (y + *D.33823);
  j = j + 1;
  if (this->dims[1] > j) goto <L1>; else goto <L3>;

<L3>:;
  i = i + 1;
  if (this->dims[0] > i) goto <L2>; else goto <L5>;
<L2>:;
  if (this->dims[1] > 0) goto <L22>; else goto <L3>;
```

# Appendix D

# Contents of the Supplemental CD

This thesis is accompanied by a CD which contains the following items:

- Patch applicable to gcc main development tree (made on revision 123774) which itself constitutes the entire implementation of interprocedural propagation of constants within aggregates. It's file name is ipa-agg-cp.diff.
- The main file of the pass for more convenient reading. The file name is ipa-agg-cp.c.
- PDF version of this thesis.
- Both examples of implementation of array expressions.

# Bibliography


[1] Fortran 77 standard. http://www.fortran.com/fortran/F77_std/rjcnf0001.html.

[2] Freepooma.

[3] GNU Compiler Collection online documentation. http://gcc.gnu.org/onlinedocs/.

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

[5] Fortran faq. http://www.faqs.org/faqs/fortran-faq/, January 1997.

[6] 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.

[7] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, 1994.

[8] Thomas Ball and James R. Larus. Branch prediction for free. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 300–313, 1993.

[9] F. Bassetti, K. Davis, and D. Quinlan. Toward fortran 77 performance from objectoriented scientific frameworks, 1998.

[10] 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.

[11] Francois Bodin, Peter Beckman, Dennis Gannon, Jacob Gotwals, Srinivas Narayana, Suresh Srinivas, and Beata Winnicka. Sage++: An object-oriented toolkit and class library for building fortran and C++ restructuring tools. In *Proceedings. OONSKI '94*, Oregon, 1994.

[12] 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.

[13] 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.

[14] Dhruva R. Chakrabarti and Shin-Ming Liu. Inline analysis: Beyond selection heuristics. In *CGO '06: Proceedings of the International Symposium on Code Generation and Optimization*, pages 221–232, Washington, DC, USA, 2006. IEEE Computer Society.

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

[16] C Standards Committee. Standard for programming language C99.

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

[18] Keith D. Cooper, Mary W. Hall, and Linda Torczon. Unexpected side effects of inline substitution: a case study. *ACM Lett. Program. Lang. Syst.*, 1(1):22–32, 1992.

[19] Dibyendu Das. Function inlining versus function cloning. *SIGPLAN Not.*, 38(4):18–24, 2003.

[20] 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.

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

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

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

[24] IBM. IBM AIX compiler information center. http://publib.boulder.ibm.com/infocenter/comphelp/v7v91/index.jsp.

[25] Steve Karmesin, James Crotinger, Julian Cummings, Scott Haney, William J. Humphrey, John Reynders, Stephen Smith, and Timothy Williams. Array design and expression evaluation in pooma ii. In *ISCOPE '98: Proceedings of the Second International Symposium on Computing in Object-Oriented Parallel Environments*, pages 231–238, London, UK, 1998. Springer-Verlag.

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

[27] 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.

[28] Wei Li and Keshav Pingali. A singular loop transformation framework based on nonsingular matrices. In *1992 Workshop on Languages and Compilers for Parallel Computing*, number 757, pages 391–405, New Haven, Conn., 1992. Berlin: Springer Verlag.

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

[30] Steven S. Muchnick. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1997.

[31] John Prentice. FORTRAN 90 vs C++ – an educational perspective. http://www.cts.com.au/compare.html.

[32] John V. W. Reynders, Paul J. Hinker, Julian C. Cummings, Susan R. Atlas, Subhankar Banerjee, William F. Humphrey, Steve R. Karmesin, Katarzyna Keahey, Marikani Srikant, and Mary Dell Tholburn. POOMA: A Framework for Scientific Simulations of Paralllel Architectures. In Gregory V. Wilson and Paul Lu, editors, *Parallel Programming in C++*, pages 547–588. MIT Press, 1996.

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

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

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

[36] 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.

[37] 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.

[38] Peter Wegner. Concepts and paradigms of object-oriented programming. *SIGPLAN OOPS Mess.*, 1(1):7–87, 1990.