Impact of Economics on Compiler Optimization

Arch D. Robison
KAI Software, a Division of Intel Americas, Inc.
1906 Fox Drive
Champaign IL 61820-7345
(217)-356-2288
arch.robison@intel.com

ABSTRACT

Compile-time program optimizations are similar to poetry: more are written than are actually published in commercial compilers. Hard economic reality is that many interesting optimizations have too narrow an audience to justify their cost in a general-purpose compiler, and custom compilers are too expensive to write. An alternative is to allow programmers to define their own compile-time optimizations. This has already happened accidentally for C++, albeit imperfectly, in the form of template metaprogramming. This paper surveys the problems, the accidental success, and what directions future research might take to circumvent current economic limitations of monolithic compilers.

Categories and Subject Descriptors
D.3.4 [**Programming Languages**]: Compilers, Optimization.

General Terms
Performance, Economics, Languages.

Keywords
Compilers, Economics, Optimization.

1. INTRODUCTION

Remarkably little is said these days about the economics of software compilers. The author searched the INSPEC database for articles written between 1969 and Jan. 2001 for the terms "compiler" and "economics". The results are illustrated in Figure 1. Most of the articles about how economics impacts compilers, however, concern "*silicon compilers*" for VLSI design, and whether they really contend with the manual alternative. Only one paper was found that really addressed the economics of compilers: a retrospective [2] on the development of the FORTRAN language. Since then, it seems, compilers have been taken for granted.

This paper resurrects the economics question, and how it impacts program optimization in the real world. This is not a research paper. It is experience of someone who has been, for 7 years, the lead developer for a commercial compiler product, which first implemented some optimizations [22][23] essential to high-performance C++ [21]. I have watched how hard it is in practice to sell compilers, and how academic priorities on compilers, notably on "optimization", depart from commercial ones.

The purpose of this paper is to bring some issues to attention of researchers, and in particular, the need to break down the monolithic structure of compilers so that boutique optimizations can be separated from the rest of a compiler. It is of particular relevance to high-performance object-oriented and parallel programming, because these domains increasingly depend upon sophisticated compiler optimizations. Those optimizations are not just a matter of technology, but also a matter of market forces.

This paper does *not* call for exclusively practical compiler research. Concerns of commercial viability should *not* limit academic research. The focus of this paper is small-market optimizations, and possible research avenues for pushing them around current commercial barriers, into the hands of users who want them.

Section 2 gives an overview of a compiler product. Section 3 considers why people buy compilers, and non-compiler competitors in the "market" for optimization. Section 4 considers the "accidental success" of template metaprogramming. Section 5 enumerates how "optimizers" might be separated from the rest of a compiler. Section 6 is a summary.

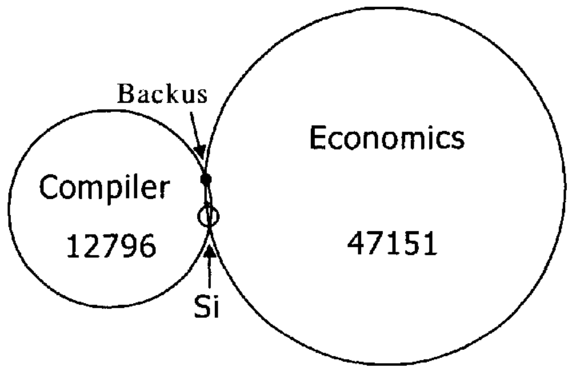


Figure 1: Compilers and economics

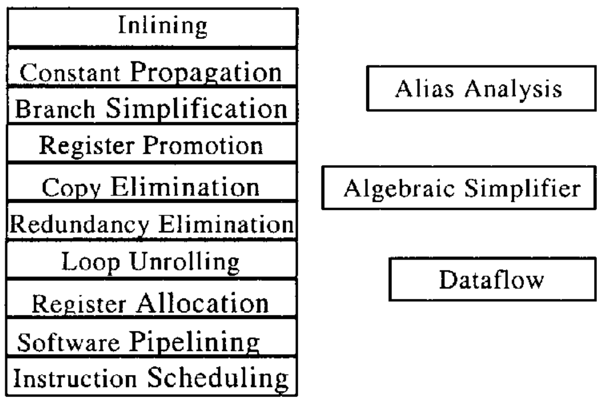


Figure 2: Register promotion is one part of optimizer.

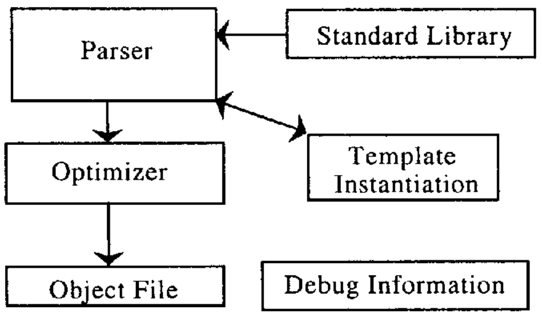


Figure 3: Optimizer is one part of compiler.

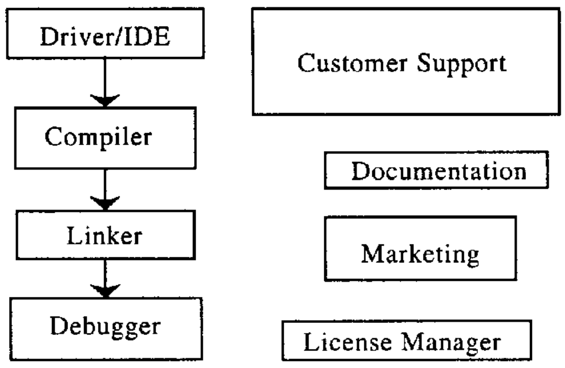


Figure 4: Compiler is one part of product.

2. PRODUCTION COMPILERS

Introductory compiler courses frequently concentrate on parsing issues. Usually, there is a later course on compiler "optimization", the traditional misnomer for "performance improvement some of the time". Many papers are published on optimization issues, probably for the reason that optimization techniques are intellectually challenging, and there are quantitative ways to measure results. Also, I suspect, it appeals to a human fascination with speed and the one-upmanship of an academic stock car race – who can make a benchmark faster?

Optimization is a fascinating issue. One of my favorite issues is register promotion, which changes operations on memory locations to operations on registers. It is crucial to high performance C++, because without it, small objects end up in memory instead of registers. There has been much research on the subject within the last five years [7][17][24]. Figure 2 sketches how register promotion is one component of many in an optimizer. Each component is quite complex – entire theses have been written about some if not all of these components. But this is just a start. Figure 3 zooms out to show how the optimizer is one component of many in a compiler. Despite the academic emphasis on parser and optimizer issues in compilers, libraries can consume surprising amounts of development resources. For instance, the ISO C++ standard devotes approximately equal number of pages to the core language versus the standard library. A commercially important point is that at this level, the components can be bought from other parties. For example, the KAI C++ compiler uses a parser from Edison Design Group (EDG) and a library originating from Modena. By generating C intermediate code, it indirectly uses the object file tools supplied by the platform's C compiler. Vice-versa, KAI sells its optimizer technology to other companies.

Despite its complexity, the compiler in Figure 3 is *not* a saleable product. When a customer buys a "compiler", they expect a lot more: drivers, linkers, debuggers, etc., as shown in Figure 4. All of these components vie with the optimizer for company resources. In a production compiler, a specific optimization such as register promotion is a tiny part, and companies, at least those who sell compilers for profit, cannot afford to focus excessively on it.

3. COMPILER MARKET

All commercial compilers need selling points. The most elementary is that the compiler handles a desired source language and target platform. Beyond that, there are product differentiators such as:

(a) Standards conformance – modern languages, notably C++, have become so complicated that some programmers are overjoyed to find a compiler that correctly parses their linguistic creations.

(b) Customer support – this includes not only fixing problems in the product. Languages like C++ are so complex that users are grateful for expert advice on why the language does not work as they expect. In many ways, compiler companies really provide not a product, but a service of enabling program development. This is exemplified in the extreme by companies that give away the compiler, but charge for maintenance and consulting.

(c) Debugging – delivering slow software on time beats delivering fast software late. Purify [14] is a striking example: its success rests on a performance *pessimization* that greatly lowers performance, but is invaluable in its ability to track down programmer errors.

(d) Compilation speed – programmers spend more time compiling codes than running them. Borland rose from obscurity with a blindingly fast Pascal compiler.

(e) Price – a major point in favor of the GNU C compiler. The Free Software Foundation's definition of "free" technically refers to intellectual property issues, but de facto the definition is "no charge" for many. However, since other

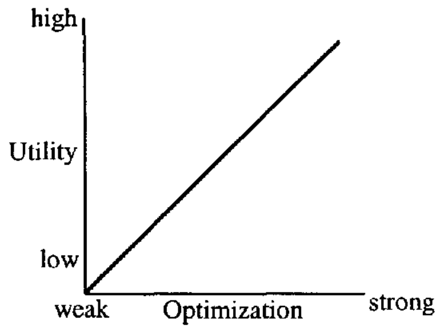


Figure 5: Possible utility function for user of a compiler

compiler vendors remain viable, it's clear that price isn't everything.

(f) Development environment – creature comforts count for a lot to some programmers. Indeed, the development environment is the centerpiece of some compilers.

(g) Optimization – performance of the generated code is important to some buyers.

Initial development of the KAI C++ compiler concentrated on (g). We were sure it was what people were looking for, largely since the earlier KAP optimizer for FORTRAN had been such a success. To our chagrin, we learned that people were primarily buying our compiler not for (g), but mostly for (a) and (b). Feature (a) was mostly luck: our optimizer happened to be designed in a way to allow new front-ends from EDG to be put in the field quickly. Optimization is just one of many differentiators. It's dear to some customers, but alone does not make a living.

3.1 Utility Functions

Obviously, different specific customers may have differing priorities. A fundamental economic notion is that of utility (how much satisfaction a consumer derives from a choice). Figure 5 shows a possible graph of compiler utility as a function of its optimization ability. As with many graphs in economics textbooks, the units and exactness of the graph are irrelevant: the point is that the slope is upwards. Increasing optimization yields increasing utility. This is true for some users. For example, a user spending CPU-months on a supercomputer can greatly benefit from a better optimizer. For example, the author knows of one case where a better optimizer for one supercomputer improved its throughput so much that it permitted shortening the lease on another supercomputer.

But the smooth upward slope is not representative of all buyers. Figure 6 shows two other possible utility functions common in practice. Graph (a) in fact, shows the most common situation for computer users. It is for users who get acceptable performance from a relatively weak optimizer, or *users who do not have the source code*. It is easy to forget this in graduate school, where source code is free and programmers abound. But in the world at large, few people can recompile a program. This detail alone limits the size of the market for exotic optimizing compilers compared, for instance, to video games. (But this does offer hope that there might be a larger market for run-time optimization than for traditional compile-time optimization.)

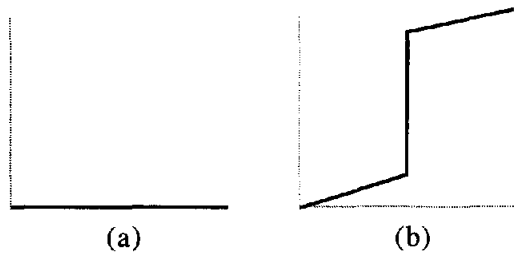


Figure 6: Other possible utility functions.

Not all is doom. Graph (b) shows another situation, and is the one that pays for optimizers: where there is a steep step upwards after a certain level of optimization is reached. It arises because the compiler won a race. For instance, perhaps it allowed embedded-processor vendor X to get better benchmark numbers than vendor Y, and consequently X gets a large contract. This explains why hardware companies can ship compilers at a loss, because they are a component of a profitable system. Purely software companies that must profit on the compiler must be careful how they compete against such subsidized compilers.

Another case where graph (b) applies is when the alternative is another programming language. For example, a good C optimizer can convince embedded programmers to switch from assembly code. A good C++ optimizer can convince some scientific programmers to switch from FORTRAN. In both cases, the programmer demands a certain level of performance, and until it is reached, has no interest.

But a buyer of a compiler has other ways to spend resources to make a program run faster. These compete against implementers of optimizers for support (money) from customers:

(a) Add more memory.

(b) Upgrade to a faster processor.

(c) Use multiple processors.

(d) Use a different algorithm.

Indeed, (a) and (b) are often the most effective. They are "off the shelf" solutions requiring little expertise, indeed nothing more than cash. They do not require the source code. Options (c) and (d) require significantly more technical expertise. There is eternal hope that (c) will eventually be done automatically *and widely*. The utility of such an optimizer would be high. It remains a holy grail. Option (d) is a frequent suggestion, and can be staggeringly effective. But the algorithm may already be the best known, and integrating a better algorithm into a program usually requires the source code and detailed understanding of how it works. Or, the opportunity for improvement may be distributed throughout the code, thus requiring extensive rewriting.

3.2 Free Compilers

Proponents of free software are by now shouting that the root problem is proprietary compilers, and that GCC is the solution. Programmers who want custom optimizations are free to modify GCC's source. This is a solution, on occasion. I've done it myself, once adding a peephole optimization to GCC for sake of a production seismic-imaging application. But doing so depends upon knowing how to write the source code in a way that triggers the peephole optimization, and locks one into a particular compiler. Furthermore, modifying a compiler requires high technical expertise that often would mean having to hire consultants. A scan of GNU-for-hire consultants shows they are most definitely not "free", which brings us right back to the market issues.

3.3 Bottom Line

The bottom line is that compilers are complicated programs that are expensive to build. Optimizations play a minor part, because they matter to only part of the compiler market. The distressing consequence is that small programming niches that would greatly benefit from certain optimizations cannot get those optimizations implemented, because the payback for the compiler vendor is not there.

Science depends upon predictions that might be confirmed or falsified in the future. Here's one for the outlined economic theory:

> For code generated by mass-market compilers, VLIW machines will be no faster than their contemporary CISC/RISC counterparts.

To make "mass market" quantitative, define it to be the top five selling compilers. The prediction is based on the assumption that VLIW machines intrinsically depend upon more exotic (and thus expensive) compilation technology to gain their advantage, and that mass-market compilers will devote their energies to other issues (such as the latest programming toolbox or debugging gizmos). The exotic optimizations will be the preserve of hardware vendors and boutique compiler companies. E.g., the Intel Pentium 4 Processor Optimization Manual [15] quite strongly recommends using the Intel Reference Compiler.

4. THE ACCIDENTAL SUCCESS

The root problem is that compile-time optimizers are an integral part of the compiler, not extensible by the programmer. Application programmers usually cannot extend them to meet the needs of their niche. What's needed is to separate the optimizer from the rest of the compiler.

There has been one real success in practice so far in this area: template metaprogramming. There are other potential successes, to be discussed in Section 5. This section summarizes why template metaprogramming is a success, and why the success is only partial.

Template metaprograms are programs that run at compilation time. When the ISO C++ committee added templates to C++, the following features were required:

- Instantiation of member types (and member constants of integral type) is demand driven.
- Template identifiers may be overloaded, with resolution based on pattern matching.
- Templates may be recursive.

The list above makes obvious to functional programmers what the committee did not realize until later [29]: templates are a functional language, *evaluated at compilation time.*

What is not obvious is that these programs can operate on non-trivial data structures such as trees. For example, Figure 7 shows a sample tree with integers on its leaves, and a hypothetical functional programming fragment for summing the leaves. Figure

```
Sum(Cons(H,T)) = Sum(H)+Sum(T)
Sum(Int(I)) = I
...
x = Cons(Cons(Int(2), Int(3)), Int(5))
... = Sum(x)
```

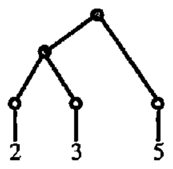


Figure 7: Functional program for summing leaves of tree

8 shows the corresponding C++ program. The C++ program is peculiar in that it represents the tree by *types*, not values. The program is syntactically ugly, but functional in every sense of the word. It is evaluated at compilation time: `sum<x>::result` can be used where an integer constant is required by C++ (e.g. case labels and in declarations of bit fields). The compile-time portion of the program is more or less isolated from the run-time portion by angle brackets. This stratification is relevant to the discussion in Section 5.

4.1 Power

The power of template metaprogramming is that it allows clever programmers to write their own compile-time optimizations, by encoding parse trees of expressions as types, and operating on them during compilation. Space does not permit explaining the technique in detail. Interested readers should consult references [13] and [26].

Despite being accidental, template metaprogramming is surprisingly powerful, for the following reasons:

(a) It uses standard ISO C++ features. It does not depend upon language extensions, or exotic compiler optimizations, though a few of the latter permit simpler templates.

(b) It has some cross-module optimization power, once C++ compilers implement the ISO C++ keyword "export" that causes the use of a template in one module to trigger evaluation of a template in another module.

(c) It is an integral part of the language with well-defined semantics. There have been proposals in the past for elaborate macro processors/editors that would optimize code, but these were tacked on top of an existing language.

(d) Powerful domain-specific optimizations are possible (e.g., a substitute for loop fusion for the loops implicit in vector expressions[13][26]).

```
template<typename H, typename T> struct Cons { };
template<int I> struct Int { };
template<typename X> struct Sum;
template<typename H, typename T> struct Sum<Cons<H,T> >
{
   static const int result = Sum<H>::result + Sum<T>::result;
};
template<int I> struct Sum<Int<I> >
{
   static const int result = I;
};
typedef Cons<Cons<Int<2>, Int<3> >, Int<5> > x;
... Sum<x>::result ...
```

Figure 8: Template metaprogram corresponding to Figure 7.

Indeed, the combination of template metaprogramming and a compiler with some optimizations for cleaning up afterward can yield code as good as FORTRAN, when applicable.

The technique is used in real production codes, which separates it from the research directions to be discussed in Section 5. Real world uses of template metaprogramming include the Blanca project at Los Alamos, a commercial finite-element package (www.cervenka.cz), a commercial "special effects" program (Avid Technology's *Elastic Reality 3.0*), simulation of vortex collisions[3], and a commercial search engine (http://www.echo.fr).

The irony is twofold: programmers using functional programming in an imperative language, and for sake of *high performance*, contrary to the low-performance caricature of functional programming languages.

4.2 Limitations

Template metaprogramming is no panacea. It suffers serious limitations. First, the designers of C++ and compilers thereof never anticipated it. Consequently, template metaprograms tend to be syntactically obtuse, and often compile quite slowly. The latter occurs where the parsers use algorithms designed for "small N" with bad asymptotic performance, say the number of instances of a template, and the metaprograms cause N to be quite big. The designers of C++ parsers never intended for them to be exploited as language interpreters. It's not clear if the template metaprogramming market is big enough to encourage parser vendors to improve the situation.

Second, and more importantly, template metaprograms "cannot see across a semicolon". They operate on individual expressions, which can even extend across modules via inlining. But they do not operate across multiple statements. For example, they are quite effective for optimizing multiple vector operations within the same expression, but do nothing for two related expressions in different statements.

4.3 Conclusion

Template metaprogramming is used in practice because it is effective in some situations and requires no language extensions. But it is an awkward "Panda's Thumb" [11] in the evolution of C++. Its fundamental limitation is its myopic scope of program analysis. Its big contribution is that it demonstrates that portions of compile-time optimization *can* be separated from a compiler, and used to benefit real codes. The next section surveys what research directions might be profitable in breaking compile-time optimizers away from the rest of the compiler.

5. RESEARCH DIRECTIONS

This section considers possible research directions in how compile-time optimizations might be separated from the compiler in a way that lets applications writers add new optimizations. The general notion is not new. The author remembers Ian Angus proposing it [1] at the Concurrent Objects Workshop at OOPSLA '93. What is new here is a deeper analysis of some of the possible options.

5.1 Desiderata

A compiler has three basic tasks with respect to an optimization:

(a) Determine if a transformation is legal.

(b) Estimate if the transformation is beneficial.

(c) Do the transformation.

The analyses for parts (a) and (b) usually get the attention, though in some cases the mechanics of (c) can be quite complicated (e.g., consider scope issues for inlining). The above are minimal criteria for an optimizer. Here are more desired traits:

(d) Allow the programmer to see some sort of semantic representation of the program before and after the transformation.

(e) Have comprehensible guidelines on the scope of its power.

(f) Have robust power.

(g) Light burden on compiler vendor.

(h) Portable – it needs to be standardized and have multiple vendors.

Trait (d) is essential, because as Angus pointed out, a third-party optimizer raises the issue of tracking down bugs. If turning on the optimizer causes a program to behave incorrectly, it may be difficult to determine who is at fault. Sometimes, the optimizer has merely exposed a latent error in the program or rest of the compiler. Compiler writers resort to displaying the internal representation before and after an incorrect transform. A user-extensible system almost surely must provide a similar facility.

An alternative to (d) would be automatic verification of user-defined optimizations. There is work [18] on such for built-in optimizations. But this is unrealistic for user-defined optimizations, since they often depend upon domain-specific knowledge.

Trait (e) is vital for practical use. A tennis racket with an unknown "sweet spot" is useless. Optimizers have their sweet spots too. A problem of many advanced optimizers (the author's included) is that a programmer is often unsure of the prerequisites for ensuring good results. A big advantage of template metaprograms is that the programmer at least knows that the templates will be instantiated at compilation time.

Trait (f) can be formalized as "how well does it tolerate trivial transformations?" For example, if the programmer rewrites "y = f(g(x))" with a temporary so it reads "{t=g(x); y=f(t);}", does the system do as well? As mentioned in Section 4, template metaprogramming is stopped cold by semicolons between statements.

Trait (g) is pragmatic. Vendors will shy away from complex behemoths. After all, the point of breaking out the optimizer is to shift the burden from the compiler vendor to third parties, not make life more miserable for the vendor.

Trait (h) comes well after issues (a)-(g) are solved. Lack of ubiquity has crushed many better mousetraps. It may sound quixotic to hope for vendor support, but it's not. OpenMP [20] is a good example of a niche language extension supported by multiple vendors.

5.2 Candidates

This section enumerates some possible candidate techniques for separating optimizers from the compiler, and comments on their likelihood for success, starting with the worst and finishing with the best.

5.2.1 Preprocessors

A popular approach is a preprocessor or postprocessor. Indeed, KAI's claim to fame was the KAP preprocessor for FORTRAN, which did automatic parallelization and cache management. On many platforms, the KAI C++ compiler is a preprocessor (from C++ to C). As an expert on this approach, I recommend *not* pursuing it, except for proof of concept. The reasons are fourfold.

First, to really work, the preprocessor has to parse the entire language. To optimize a program requires understanding its structure. The preprocessor cannot be like the C preprocessor, which is clueless about even the grammatical structure of the C language. This implies that a serious preprocessor needs an almost complete parser for the language. Regrettably, building such parsers (and unparsers) for languages has become quite an expensive task, though one can buy these components from other parties (e.g. EDG). Either way, acquiring a parser is non-trivial.

Second, a transformation by preprocessing almost surely ends up misplaced in the optimization sequence. Some optimizations need to precede others to be effective. For instance, doing inlining last is quite ineffective (though the author once saw a compiler that appeared to do exactly that!) Others need to come late; e.g., scheduling for the currently fashionable SIMD extensions. By its very nature, a preprocessor does optimizations before the rest of the compiler. A postprocessor is the right answer in some cases (e.g. Spike [6]). But usually, optimizations are best done at a specific point in the middle of the optimization sequence where they have maximal impact.

Third, it may be *impossible* to express the optimization in the source language. For example, suppose we want to build a C++ to C++ preprocessor that does inlining. Consider the code in Figure 9. When control leaves the scope of variable x, the destructor ~Foo() must be called. Can a preprocessor inline the destructor at the call site? The answer is no, for two reasons. First, the variable n is private; it can be accessed only by methods of class Foo. This could be worked around by having the preprocessor insert public declarations. This is a common disease of preprocessors -- extra transformations for sake of making the code palatable to the compiler. But there is a second, more insidious problem in the example: the call site to the destructor is really white space, and cannot be removed by the preprocessor! Figure 10 shows three attempts to do the inlining. Attempt (a) inserts "++n" at the call site, but this causes n to be incremented twice: once by the inserted code and once by the implicit call to the destructor. Attempt (b) tries to avoid this by removing the declaration of x, but that omits the required implicit call to the constructor Foo(). Attempt (c) circumvents this with a call to Foo(), but the rules of C++ syntax cause that to be treated as construction of *another* object (and its destructor will increment n again). Trying to work around these problems is a fun exercise, but digs a yet deeper grave of more complicated transformations.

```
class Foo {            |
private:               |
   static int n;       |   {
public:                |      Foo x;
   Foo();              |   }
   ~Foo() {++n;}       |
};                     |
```

Figure 9: C++ source-level inlining problem.

```
{                |  {              |  {
   Foo x;        |     ++Foo::n;   |     Foo();
   ++Foo::n;     |  }              |     ++Foo::n;
}                |                 |  }
      (a)        |       (b)       |       (c)
```

Figure 10: Three incorrect inlinings.

Fourth, the preprocessor approach is just plain unwieldy. Imagine having to run *N* preprocessors because you are using *N* libraries. Not only is there a lot of parsing/unparsing going on, but also each preprocessor is likely to be clueless about what the other preprocessor is doing. For instance, an analysis might require a fixpoint solution to a problem concerning multiple libraries. Surely, a more integrated approach is desired.

5.2.2 Plug-in Optimizers

Web browsers have plug-ins, often implemented as dynamically shared libraries. So why not do the same for a compiler? The compiler could load a new pass written as a dynamic shared library. Optimizations written by third parties could place themselves appropriately in the optimization sequence, and benefit from all the utilities in the compilers (e.g. alias analysis). This works sometimes. For instance, the optimizer in development versions of KAI C++ is a dynamically shared library. Each pass dynamically registers itself at startup.

A research implementation of a plug-in compiler is the MAGIK system [10]. It allows users to extend the compiler via dynamically shared libraries. The extensions inspect and transform the low-level tree intermediate representation used by the compiler.

The drawbacks to a plug-in optimizer are:

(a) It requires detailed knowledge of the compiler's internal representation of a program, which is often strewn with decorations that must be maintained for sake of other optimizations.

(b) Compiler internal representations are often short lived, and peculiar to particular target platforms. The lifetime of a typical internal representation is a decade or two, and undergoes many revisions over the lifetime.

(c) It locks the application program into a particular compiler. (This is a drawback for the application writer, not for the compiler vendor.) Of course, some application writers lock their programs into a vendor's compiler anyway, by using the vendor's language extensions, so this objection is not necessarily prohibitive.

The above objections also apply to "plugging in" by modifying the source to a compiler, regardless of whether the source is freely available or bought.

5.2.3 *Reflection and Metaobject Protocols*

Reflection is the ability of a program to look at itself. It's worth thinking about how a program might look at itself and optimize itself. It is conceivable that a system of reflection could allow a library to inspect its compile-time representation and modify itself at compile time. However, this is the sort of thing that scares off traditional optimizers, so it would need to be tamed somehow, in a way that allows optimizers to work in conjunction with it. Perhaps some sort of stratification would work, analogous to the way template metaprogramming uses <> to delimit the world of compile-time calculations from run-time calculations.

A formal object-oriented approach to reflection is the notion of metaobject protocols. Lamping et al [16] propose opening up a compiler by letting the user specify compile-time metaobject protocols that assist implementation decisions by the compiler. OpenC++ [4][5] implements compile-time metaobject protocols for C++. Each object in the user's program can have a corresponding "metaobject" with methods that inspect and modify the program's parse tree. Thus, OpenC++ is truly an extensible compiler. The drawback is the dependence upon a particular parse-tree dialect. It ties metaprograms to a specific internal representation of a program, just as a plug-in optimizer would. The candidates to be introduced below minimize or eliminate the dependence on parse trees by giving up on full generality.

5.2.4 *Partial Evaluation*

Partial evaluation attempts to evaluate those parts of a program that do not depend upon run-time data values. This may involve specialization and cloning of routines and data structures for specific instances where only part of the instance is known at compile time.

Template metaprogramming is a form of partial evaluation [27], where values that would be known at compile time are encoded as types. Veldhuizen [28] points out that techniques similar to template metaprogramming could be employed in Java, given a compiler with sufficiently powerful partial evaluation.

Partial evaluation can be leveraged for user-defined optimizations by adding a simple predicate "true_at_compile_time(*p*)", which would return true if and only if the value of *p* is known to be true a compilation time. It would be used like this:

```
bool p = gather information about aspect of program
if( true_at_compile_time(p) ) {
  use specialized form
} else {
  use general form
}
```

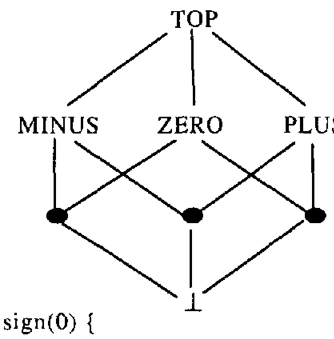
The notion is that "gather information about program " might be too expensive to leave as a run-time check. Thus at compilation-time, partial evaluation would try to prove that *p* is always true, and the branch could be simplified to either the special or the general case. The assignment to *p* would become a dead store to be removed. Clever programmers already use this technique when they know that a compiler will determine the truth/falsity of *p* at compilation time. The predicate "true_at_compile_time" suggested here allows programmers to extend the trick to situations where falsity is not always certain.

Partial evaluation avoids introducing a meta-level language. This brings strength of simplicity, but also weakness of uncertainty. Without delineation between compile-time and run-time calculations, the programmer must understand the abilities and limits of the partial evaluator. Otherwise, code intended for compile-time evaluation may end up as run-time code, introducing overhead instead of eliminating it. Two-level functional languages [19] and multistage programming [25] attack this drawback by letting programmers explicitly mark code that must be evaluated at compile time. It remains to be seen if this sort of marking can be made practical for imperative real-world languages.

5.2.5 *User-Level Abstract Interpretation*

A key element of static analysis is computing a conservative approximation of a program's behavior, and choosing transformations based on the approximation. The underlying theory is known as abstract interpretation [8], in which the program is interpreted with respect to an artificial lattice domain. It seems natural to find some way to let users define their own approximations this way.

One way to do this might be to introduce a notion of "lexical instance variables". The term is based on Smalltalk terminology. An ordinary "instance variable" is one for which there is a



```
class Real {
  enum sign_t {
    TOP=0,
    MINUS=1,
    ZERO=2,
    PLUS=4
  };
  float value;
  lexical sign_t sign;
public:
  Real() : sign(TOP) {}
  Real( float x ) : value(x), sign(0) {
    if( x<0 ) sign|=MINUS;
    if( x>0 ) sign|=PLUS;
    if( x==0 ) sign|=ZERO;
  }
  Real operator=( const Real& y ) {
    value = y.value;
    sign = (sign_t)(sign | y.sign);
  }
  friend Real operator*( const Real& x, const Real& y ) {
    Real z;
    z.value = x.value*y.value;
    if (x.sign&ZERO || y.sign&ZERO)  z.sign|=ZERO;
    if ((x.sign&MINUS) && (y.sign&PLUS)) z.sign|=MINUS;
    if ((x.sign&PLUS) && (y.sign&MINUS)) z.sign|=MINUS;
    if ((x.sign&PLUS) && (y.sign&PLUS)) z.sign|=PLUS;
    if ((x.sign&MINUS && y.sign&MINUS)) z.sign|=PLUS;
    return z;
  }
  friend bool operator<( const Real& x, const Real& y ) {
    if( (x.sign & y.sign)==0 )
       return x.sign<y.sign;      // Have compile-time answer
    return x.value<y.value;
  }
}
```

Figure 11: User-defined abstract interpretation for "rule of signs" via hypothetical minimalist language extension.

separate instance per each instance of the containing class. These are called "fields" in C++. A "class variable" is one for which there is one shared by all instances. These are called "static data members" in C++. The proposal here is for a third category: one per lexical occurrence of an instance of the class, or one per instance of the class in the compiler's internal representation. The latter differs from the former if the compiler indulges in any code replication, e.g. inlining.

These lexical instance variables would give users power analogous to decorating the parse tree. If these variables were considered strictly compile-time entities, they could be used to compute user-defined attributes. If suitable stratification restrictions were made (including monotonicity constraints etc.) in a way that lexical instance variables could be computed at compilation time (by some fixpoint rule that dealt with control-flow interpretation), then programmers could use lexical instance variables to specify their own dataflow analyses using the usual source language data types and operations, and take action on the results using standard conditional logic.

Figure 11 shows how this might work for the "rule of signs" [8] for a class representing real numbers. The keyword **lexical** marks field "sign" as a lexical instance variable. It would exist only at compilation time. The lattice for the abstract interpretation is represented as a three-bit vector. The lattice meet operation is bitwise-OR. Various operators are overloaded to compute both their run-time results, and their compile-time abstract interpretation. The example presumes some partial evaluation capability on part of the compiler, in order to conservatively track whether each bit in the "sign" field is zero, one, undefined, or indeterminate. This use of bit vectors may seem a bit ad-hoc to readers unfamiliar with the internals of compilers. However, production compilers often represent lattice values by bit vectors, because any finite lattice is embeddable on a hypercube of sufficient dimension. Thus, support in the compiler for lexical instance variables, and partial evaluation of bit vectors, along with constant propagation, might actually suffice for many user-defined optimizations. This is just a suggested line of attack. It is only a fuzzy sketch. But it shows how one could go about minimizing language extensions by allowing the programmer to use mostly existing operators and data types.

A "lexical instance variable" is a compile-time metaobject. The metaobject protocol here is quite constrained. Unlike the protocols discussed in Section 5.2.3, it is not stated directly in terms of parse trees. There are no explicit references to child or parent parse nodes. The lexical instances, of course, correspond to some sort of internal representation, but the communication between instances is formulated in terms of the user-level program, not parse-tree links. Instead of modifying parse trees, the lexical instances merely provide compile-time approximations of run-time behavior. Ordinary branch simplification does the actual program transformation by choosing between alternative implementations. Avoidance of explicit parse trees avoids tying the program to a particular compiler's internal representation. The constraints buy simplicity by giving away generality. Determining whether this trade is practical is an open question, resolvable only by real experience.

5.2.6 *User-Defined Dataflow*

Dataflow analysis is the most common form of abstract interpretation in a compiler. So why not let the user extend it? The Broadway Compiler [12] does this. It is easily the most ambitious and advanced attempt at breaking optimizations out of the compiler. It is motivated by the need for domain-specific optimizations. Library implementers use an annotation language to specify their own dataflow analyses and transformations based on those analyses, typically specializations. The work looks promising, because it goes well beyond the simple pattern matching semantics of previous annotation-based systems.

A minor drawback of the system is that its annotation language is a separate language. Its user does not have access to a full-blown programming language. E.g., the system operates on C programs, but the user defines the analyses in a language that is not C.

5.3 Practicality

The candidates of the preceding section all require considerable sophistication on the part of the programmer, so it is fair to ask if they are too complicated for real production users. Perhaps so, but there are two points for hope. First, expression templates are not easy to understand either, yet they are in use, proving that users desperate for high performance are willing to learn. Second, it's not the "average user" who needs to understand them, but the high-performance library writer who does. Users of template-expression packages often do not understand template metaprogramming. Numerical programmers regularly call BLAS libraries written by experts in cache management. If library writers can learn the arcanum of templates and cache hardware, they can similarly learn the arcanum of a little applied lattice theory.

5.4 Definitive Problems

Evaluating a particular proposal to break an optimizer out of a compiler requires some definitive and useful optimization problems to test it on. Here are some that come to mind:

(a) Parallel programming – does the system enable (or at least assist) automatic parallelization (by library writers) of applications using their library?

(b) Error checking - does the system allow library writers to do compile-time checking of problems that otherwise would go unnoticed until run-time?

(c) Dead operation elimination - does the system allow useless library operations to be removed?

(d) Loop fusion - can the system fuse adjacent loops over the same iteration space? Of course, existing compilers can do this for trivial domains such as unaliased arrays. But what about linked lists or trees?

(e) Object inlining – does the system allow a library to perform object-inlining [9]?

(f) Garbage collection – does the system allow garbage collection to be written as a third-party library? Reference counting is often used in C++ applications instead of other methods such as stop-and-copy, because non-conservative versions of the latter require information about what is a pointer and what is not.

(g) Serialization – does the system allow the whole notion of serialization to be encapsulated in a library?

(h) C++ library – does the system allow programmers to write optimizations for C++ library functions. For example, optimize sequences of dynamic string operations. E.g., std::string() + "abc" + "efg" can, in principle, be reduced to a compile-time construction of the result.

The last item might be enough to convince compiler vendors to adopt the system, as many would probably like to farm out the library optimization issues to third parties, as they already do for the libraries. Also, if the system does well only for scientific programming, it is probably doomed. (Microsoft sold off its F90 compiler for a reason!) It will need to demonstrate significant benefits to mainstream programmers, or at least those who will pay the cost of it.

6. SUMMARY

Compiler optimization is treated in academia as a purely technical exercise. But in the real world, compiler optimizations are a matter of technology *plus market forces*. There exist competing alternatives for making a program run faster. Compilers are solid monoliths, inextensible by application programmers. Template metaprogramming is an exciting yet awkward demonstration of what application programmers can do with compile-time optimizations. There is a variety of possibilities to consider. The winners will assuredly allow the user to extend both the analysis and the transformation capabilities of the compiler, with global scope similar to that for built-in analyses. The hard part is doing this in a way that does not significantly increase the burden on the compiler vendor, for whom optimization is but a small part of a whole product.

7. ACKNOWLEDGEMENTS

This paper would not have been written without the encouragement of John Reynders. Pierre Aubert, Jan Cervenka, Garth Dickie, John Reynders, and Mathias Troyer furnished information on the real-world uses of template metaprogramming mentioned in Section 4.1. Todd Veldhuizen provided many comments and corrections. The anonymous referees contributed useful criticism of an earlier draft, and pointed out the work on compile-time metaobject protocols.

8. REFERENCES

[1] Angus, I. G. Applications demand class-specific optimizations: The C++ compiler can do more, in *Scientific Programming* 2,4 (1993), 123-131.

[2] Backus, J. The history of Fortran I, II, and III. *IEEE Annals of the History of Computing* 20, 4, (1998), 68-78.

[3] Bou-Diab, M., Dodgson, M., and Blatter, G. Vortex collisions: crossing or recombination. To appear in *Physical Review Letters*.

[4] Chiba, S. A metaobject protocol for C++, in *Tenth Annual Conference on Object-Oriented Programming Systems, Languages and Applications,* (Austin, TX, Oct. 1995), 285-299.

[5] OpenC++ Home Page http://www.hlla.is.tsukuba.ac.jp/~chiba/openc++.html

[6] Cohn, R., Goodwin, D., Lowney, P.G., and Rubin, N. Spike: An optimizer for Alpha/NT executables. In *Proceedings of the USENIX Windows NT Workshop*, (Seattle, Washington, Aug. 1997).

[7] Cooper, K. and Lu, J. Register promotion in C programs. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation* (Las Vegas, Nevada), 308-319.

[8] Cousot, P., and Cousot, R., Abstract Interpretation: A Unified Model for Static Analysis of Programs by Construction or Approximation of Fixpoints, *in Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Los Angeles, CA, 1977), ACM Press, 238-252.

[9] Dolby, J., and Chien, A. An automatic object inlining optimization and its evaluation. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Languages and Implementation*, (Vancouver, Canada), 345-357.

[10] Engler, D. Interface compilation: steps towards compiling program interfaces as languages. In *IEEE Transactions on Software Engineering* 25,3 (May/June, 1999), 387-400.

[11] Gould, S. J. *The Panda's Thumb: More Reflections in Natural History* (1980), W. W. Norton & Company, 19-26.

[12] Guyer, S. Z., and Lin, C. An annotation language for optimizing software libraries. In *2nd Conference on Domain Specific Languages* (October. 3-5, 1999), SIGPLAN Notices 35,1, ACM Press, 39-52.

[13] Haney, S., Crotinger, J., Karmesin, S., and Smith, S. PETE: The portable expression template engine. In *Dr. Dobb's Journal* (October 1999).

[14] Hastings, R. and Joyce, B. Purify: fast detection of memory leaks and access errors. In *Proceedings of Winter 1992 USENIX Conference* (San Francisco, CA, 1991), 125-138.

[15] *Intel® Pentium® 4 Processor Optimization Reference Manual*, Chapter 2, (November 2000), Intel Corporation, 7.

[16] Lamping, J., Kiczales, G., Rodriguez, L.H., and Ruf, Erik. An Architecture for An Open Compiler. In *Proceedings of the IMSA '92 Workshop on Reflection and Meta-level Architectures*, (Tokyo, Japan, 1992), 95-106.

[17] Lo, R. and Chow, F., and Kennedy, R., and Liu, S.M., and Tu, P. Register promotion by sparse partial redundancy elimination of loads and stores, in *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montreal, Canada), 26-37.

[18] Necula, G. C. Translation validation for an optimizing compiler. In *Proceedings of ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, (Vancouver, Canada), 83-94.

[19] Nielson, F. and Nielson, H.R. *Two-Level Functional Languages* (1992), Cambridge University Press.

[20] OpenMP Application Program Interface. http://www.openmp.org

[21] Robison, A. D. C++ gets faster for scientific computing. In *Computers in Physics*, 10, 5 (Sept./Oct. 1996), 458-462.

[22] Robison, A. D. Method of analyzing definitions and uses in programs with pointers and aggregates in an optimizing compiler. U.S. Patent 5,790,866 (Aug. 4, 1998)

[23] Robison, A. D. Method of replacing lvalues by variables in programs containing nested aggregates in an optimizing compiler. U.S. Patent 5,710,927 (Jan. 1998).

[24] Sastry, A. V. S., and Ju, R. D. C. A new algorithm for scalar register promotion based on SSA form. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation* (Montreal, Canada), 15-25.

[25] Taha, W. and Sheard, T. Multi-Stage Programming with Explicit Annotations. In *Proceedings of the 1997 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (Amsterdam, The Netherlands, June 1997), SIGPLAN Notices 32,12, ACM Press, 203-217.

[26] Veldhuizen, T. Expression templates. In *C++ Report* 7,5 (June 1995), 26-31.

[27] Veldhuizen, T. C++ templates as partial evaluation. In *Proceedings of the 1999 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation* (San Antonio TX, January 1999), ACM Press, 13-18.

[28] Veldhuizen, T. Just when you thought your little language was safe: Expression templates in Java. In *Generative and Component-Based Software Engineering* (Erfurt Germany, Oct. 2000).

[29] Velhuizen, T. Using C++ template metaprograms. In *C++ Report* 7,4 (May 1995), 36-43.