Minimizing Dependencies within Generic Classes for Faster and Smaller Programs

Dan Tsafrir[†] Robert W. Wisniewski[†] David F. Bacon[†] Bjarne Stroustrup[◇]

[†]IBM T.J. Watson Research Center
{dants,bobww,bacon}@us.ibm.com

[◇]Texas A&M University
bs@cs.tamu.edu

Abstract

Generic classes can be used to improve performance by allowing compile-time polymorphism. But the applicability of compile-time polymorphism is narrower than that of run-time polymorphism, and it might bloat the object code. We advocate a programming principle whereby a generic class should be implemented in a way that minimizes the dependencies between its members (nested types, methods) and its generic type parameters. Conforming to this principle (1) reduces the bloat and (2) gives rise to a previously unconceived manner of using the language that expands the applicability of compile-time polymorphism to a wider range of problems. Our contribution is thus a programming technique that generates faster and smaller programs. We apply our ideas to GCC's STL containers and iterators, and we demonstrate notable speedups and reduction in object code size (real application runs 1.2x to 2.1x faster and STL code is 1x to 25x smaller). We conclude that standard generic APIs (like STL) should be amended to reflect the proposed principle in the interest of efficiency and compactness. Such modifications will not break old code, simply increase flexibility. Our findings apply to languages like C++, C#, and D, which realize generic programming through multiple instantiations.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Polymorphism; D.3.3 [*Programming Languages*]: Data types and structures

General Terms Design, measurement, performance

Keywords Generics, templates, SCARY assignments and initializations, generalized hoisting

1. Introduction

Generic programming is supported by most contemporary programming languages [24] to achieve such goals as compile-time type safety. In languages like C++, C#, and D, generic programming also allows for improved performance through compile-time polymorphism as follows [48, 36]. Rather than generating only one version of the code (by using dynamic binding to hide the differences between type parameters), the compiler emits a different code instantiation for each new combination of the parameterized types. It is therefore able to perform static binding, which enables a host of otherwise inapplicable optimizations, notably, those based on inlining. The price is a potential increase in object code size, sometimes denoted as "bloat" [8, 29, 4].[1]

Generic classes often utilize nested types when defining their interface [9, 25]. A notable example is the iterators of STL, the ISO C++ Standard Template Library. STL is among the most widely used generic frameworks. We will use it throughout this paper to demonstrate our ideas (in Section 8 we will generalize to other libraries/languages). The iterator concept is interwoven in almost every aspect of STL.

Nested classes implicitly depend on all the generic parameters of the outer class in which they nest. Consider for example the STL sorted container std::set<T,C,A> (which stores items of the type T, compares items with a comparator of the type C, and (de)allocates memory with an allocator of the type A). If two sets agree on T but disagree on C or A, then the corresponding nested iterators are of different types. This means that the code snippet in Figure 1 does not typically compile due to type-mismatch errors.

```
set<int,C1,A1>::iterator i1;
set<int,C2,A1>::iterator i2 = i1; // different comparator
set<int,C1,A2>::iterator i3 = i1; // different allocator
```

Figure 1. *Can this code have a valid meaning? Can it be compiled by existing compilers? Can it be useful?*

And indeed, our repeated experience is that, when presented with Figure 1, well-read and experienced programmers initially react negatively and feel that this code snippet is in flagrant violation of the type system. When further presented with a "hypothetical" possibility that the snippet

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA

[1] The resulting generated code can actually be smaller than what is obtained when using dynamic binding; but this is unrelated to our definition of "bloat", which is the *increase* in size caused by *additional* instantiations.

might nevertheless compile on existing compilers, they do not understand the semantics of the code, and they fail to see why it could ever be useful.

This paper is dedicated to refuting the perception of programmers regarding Figure 1. Specifically, we show that it is possible (and rather easy) to implement the nested type (the iterator) and its encapsulating class (the container) in a way that makes Figure 1 be ISO standard conforming and accepted by existing unmodified C++ compilers. We further show that doing so is highly beneficial, because it yields a superior design that has two important advantages:

1. it emits less code when instantiating generic algorithms, and so it yields smaller executables, and
2. it allows us to write faster programs and improve the performance, by utilizing statements as in Figure 1.

Consequently, the answer to the questions raised in the caption of Figure 1 is “yes”.

1.1 Minimizing Dependencies

Let us denote assignments and initializations like those shown in Figure 1 as “SCARY assignments”.[2]

We contend that a container design that explicitly allows SCARY assignments to compile is more correct than a design that does not allow them. The well-known design principle that underlies this claim is that independent concepts should be independently represented and should be combined only when needed [49]. The inability to compile Figure 1 serves as indication that this principle was violated, because it proves that iterators depend on comparators and allocators, whereas STL iterators need not depend on comparators or allocators, as there is nothing in the ISO C++ specification that indicates otherwise.

We note that the only meaningful implication of such unwarranted dependencies is that SCARY assignments do not compile and so the aforementioned benefits (reduced bloat, better performance) are prevented. The fact that the specification of ISO C++ is silent regarding this issue (namely, it does not specify whether or not iterators should depend on comparators and allocators) attests the lack of awareness to our proposed approach and its benefits.

Technically, the unwarranted dependencies can be easily eliminated by moving the definition of the nested iterator to an external scope and replacing it with an alias to the now-external iterator; by using only `T` as its generic parameter, we eliminate the unwarranted dependencies. Doing so allows Figure 1 to compile under unmodified compilers and provides semantics to its SCARY assignments: the iterators `i1`, `i2`, and `i3` have the same type, which is a generic class that only depends on `T`. The iterators thus become *interchangeable*, regardless of the comparators and allocators utilized by the associated containers. So while the SCARY assignments appear “new” and possibly counterintuitive, there is no need to invent new semantics and to modify the compiler and language in order to make them work.

[2] The acronym SCARY describes assignments and initializations that are **S**eemingly erroneous (appearing **C**onstrained by conflicting generic parameters), but **A**ctually work with the **R**ight implementation (unconstrained b**Y** the conflict due to minimized dependencies).

1.2 Improving Performance

When programmers need to handle objects with different types in a uniform manner, they typically introduce an abstraction layer that masks the differences between the types. For example, to uniformly handle a “Circle” and a “Triangle”, we use runtime polymorphism and make them part of a class hierarchy headed by an abstract “Shape” base class.

The same technique (introducing an abstract base class) is used to handle iterators with different types in a uniform manner. But when dependencies are minimized as advocated above, the type differences may no longer exist, making iterators interchangeable and obviating the need for abstraction and runtime polymorphism. (This is analogous to discovering that “Circle” and “Triangle” actually have the same type and are in fact interchangeable.) As noted, runtime polymorphism incurs a performance penalty (e.g., hindering inlining), which is avoided if compile-time polymorphism and static binding are employed instead. This is the source of our performance improvement.

Notice, however, that the improvement is *not* merely the result of minimizing dependencies, which is necessary but insufficient for this purpose. Rather, programmers must program in a certain way: they *must* utilize SCARY assignments, as these constitute the only way by which the interchangeability can be exploited to improve the performance.

In Sections 2 and 3 we show how to solve the classical multi-index database problem without and with SCARY assignments, and we highlight the advantages of the latter approach. In Section 4 we evaluate the competing designs using microbenchmarks and a real application, and we demonstrate speedups between 1.2x to 2.1x for the application.

1.3 The Need for Standardization

Since the above benefits are nonnegligible and since obtaining them is nearly effortless, we contend that classes should be implemented to allow SCARY assignments. But this is not enough. We further contend that ability to utilize SCARY assignments should be specified as part of the API; otherwise, their use would be nonportable and might break with different or future versions of an implementation.

The general conclusion is that designers should be mindful when utilizing nested types as part of the interface. Specifically, they should aspire to minimize the dependencies between the inner classes and the type parameters, and they should specify interfaces to reflect that. This will not break existing code. Rather, it would provide programmers with the flexibility to leverage the interchangeability, and, as discuss next, it would eliminate code bloat caused by overconstrained inner classes.

#	vendor	compiler	operating system	iterator
1	Intel	C++ Compiler 11.0 Professional (ICC)	Windows	dependent
2	Microsoft	Visual C++ 2008 (VC++)	Windows	dependent
3	IBM	XL C/C++ V10.1 (xlC)	AIX	dependent
4	Sun	Sun Studio 12 C++ 5.9	OpenSolaris, Linux	dependent
5	Borland	CodeGear C++ Builder 2009	Windows	dependent
6	GNU	GCC 4.3.3	*NIX	not dependent
7	Intel	C++ Compiler 11.0 Professional (ICC)	Linux *(using the STL of GCC)*	not dependent
8	IBM	XL C/C++ V10.1 (xlC)	Linux *(using the STL of GCC)*	not dependent

Table 1. *Iterators may be declared as inner or outer, and therefore they may or may not depend on the comparator and allocator; the compiler's vendor is free to make an arbitrary decision. Until now, this has been a non-issue. (Listing includes most recent compiler versions as of Feb 2009. The "iterator" column is based on the default compilation mode. Borland has recently sold CodeGear to Embarcadero Tech.)*

1.4 Reducing Code Bloat

Replacing inner classes with aliases that minimize dependencies reduces code bloat for two reasons. First, it unifies redundant multiple instantiations of the inner classes. Without this unification, member methods of a nested iterator could be instantiated once for each comparator and allocator combination, even though all instantiations yield identical object code. The second, more important, reason is that any generic algorithm for which the inner class serves as a type parameter would, likewise, be uselessly duplicated. For example, iterators are used to parameterize most STL algorithms (e.g., std::copy, std::find, std::sort, etc.). When such an algorithm is used, any change in the iterator type will prompt another algorithm instantiation, even if the change is meaningless.

Reducing bloat by replacing inner classes with aliases can be further generalized to also apply to member methods of generic classes, which, like nested types, might uselessly depend on certain type parameters simply because they reside within a generic class's scope. (Again, causing the compiler to uselessly generate many identical or nearly-identical instantiations of the same method.) To solve this problem we propose a "generalized hoisting" design paradigm, which decomposes a generic class into a hierarchy that eliminates unneeded dependencies. We define this technique in Section 5, apply it to standard GCC/STL containers in Section 6, and show that the resulting code can be up to 25x smaller.

1.5 Generalizing

We note that generalized hoisting is not just useful to minimize dependencies between member methods and generic parameters; it can also be similarly applied as an alternative way to minimize dependencies between member classes (that is, inner classes) and generic parameters. According to this doctrine, instead of moving the iterator definition to an external scope, we could (1) define a base class for std::set<T,C,A> that is parametrized by only T, and (2) move the iterator definition, as is, to this base class. Consequently, generalized hoisting can be viewed as a generalization of our idea from Section 1.1.

1.6 Contributions and Paper Roadmap

The novelty of our work is *not* in coming up with a technical way to reduce the dependencies between inner classes and type parameters (see Table 1). Rather, it is (1) in identifying that this issue matters, (2) in recognizing that minimizing dependencies between the members and the type parameters of a generic class is a valuable design principle that can be utilized to improve performance and reduce bloat, (3) in conceiving SCARY assignments and generalized hoisting that make it possible to realize and exploit this principle, and (4) in doing the experimental work that quantifies the benefits and substantiates the case.

To summarize, our contribution is a technique that can reduce the amount of emitted generic code and make it run faster. This statement is supported by Sections 2–6 (as described above) in the context of C++. We then discuss how the compiler and language can provide support to our ideas (Section 7), we generalize our results to other programming languages (Section 8), we discuss related work (Section 9), and we conclude (Section 10).

2. Motivation

In this section we describe the problem chosen to demonstrate the benefits of the technique we propose (Section 2.1). We then describe the two standard ways to solve the problem (Sections 2.2 and 2.3). In Section 3 we will develop a third, nonstandard, solution that utilizes SCARY assignments, and we will compare it to the latter two.

The three solutions are short, which allows us to provide their full (compiling) code, promoting clarity, and, more importantly, allowing us to precisely identify the reasons for the performance benefits of our approach.

2.1 The Problem

In a nutshell, what we want is a database of items that (1) is sorted in different ways to allow for different traversal orders, and that (2) supports efficient item insertion, removal, and lookup. Numerous applications make use of such databases. For brevity, we assume that the items are integers (these may serve as "handles" to the associated objects). Let

operation	return	complexity	description
add (int i)	void	$O(K \cdot logN)$	add i to the database
del (int i)	void	$O(K \cdot logN)$	delete i from the database
begin (int k)	Iter_t	$O(1)$	return iterator to beginning of database when sorted by the k-th sorting criterion
end (int k)	Iter_t	$O(1)$	return iterator to end of database when sorted by the k-th sorting criterion
find (int k, int i)	Iter_t	$O(logN)$	return iterator to i within sequence that starts with begin(k), return end(k) if not found

Table 2. *The operations we require our database to support. The variable i denotes an item and may hold any value. The variable k denotes a sorting criteria and is in the range k = 0, 1, ... , K-1. The Iter_t type supports the standard pointer-like iterator operations.*

K denote the number of different sorting criteria, and let N denote the number of items that currently populate the database. Table 2 specifies the database operations and their required runtime complexity.

The operation add and del respectively add and delete one item to/from the database and do so in $O(K \cdot logN)$ time. The operations begin and end respectively return the beginning and end of the sequence of items that populate the database, sorted by the k-th sorting criterion. Both operations return an object of the type Iter_t, which is an iterator that supports the usual iterator interface (of primitive pointers) similarly to all the STL containers; e.g., Figure 2 shows how to print all the items ordered by the k-th sorting criterion. All the operations in Figure 2 are $O(1)$, including begin and end and the iterator operations (initialization "$=$", inequality "!=", increment "$++$", and dereference "$*$"). Consequently, the entire traversal is done in $O(N)$ time.

```
Iter_t b = db.begin(k);
Iter_t e = db.end(k);
for(Iter_t p=b; p != e; ++p) { printf("%d ", *p); }
```

Figure 2. *Iterating through the multi-index database using the k-th sorting criterion and printing all items.*

The last supported operation is find, which searches for i within the sequence of database items sorted by the k-th criterion. If i is found, the associated iterator is returned (dereferencing this iterator would yield i); otherwise, end(k) is returned. Thus, if users just want to check whether or not i is found in the database (and do not intend to use the returned iterator), they can arbitrarily use, e.g., k=0, as below:

```
if( db.find(0,i) != db.end(0) ) { /*found! ... */ }
```

(An arbitrary k can be used, because finding i in some sorted sequence means i is found in all the other sequences.) Users may alternatively be interested in the returned iterator of some specific k, e.g., if they want to examine the neighbors of i according to a specific order. The runtime complexity of find is $O(logN)$.

2.2 Using an Abstract Iterator

If the stated problem appears familiar, it is because it is similar to the problem that motivates the classic iterator design pattern as defined in the seminal work by Gamma et al. [23, pp. 257–271] and as illustrated in Figure 3. The solution is also similar. We are going to implement each sorting criterion as a container ("concrete aggregate") that is sorted differently. Naturally, we are going to use STL containers (these are readily available and provide performance similar to that of hand-specialized code), such that each container employs a different comparator. But different comparator types imply different iterator types, whereas Table 2 dictates just one iterator type for all sorting criteria. We therefore have no choice but to utilize an abstract iterator base class in order to hide the type differences as shown in Figure 3.

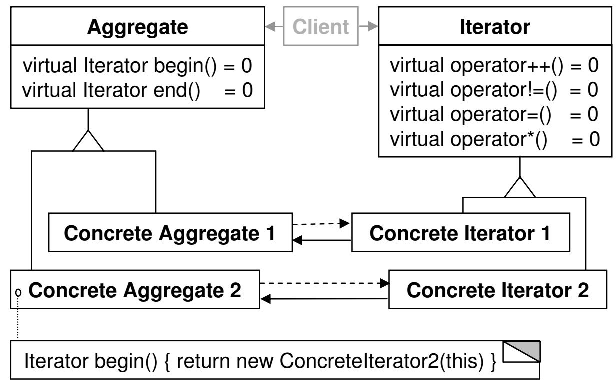

Figure 3. *The classic iterator design pattern. While the notation is adapted to match that of STL iterators, the latter do not model the classic pattern, and they have a narrower applicability.*

We stress that, contrary to common belief [17, 19, 53], C++/STL iterators do *not* model the classic design pattern. They do not involve runtime polymorphism and dynamic binding, there is no iterator base class, and different containers have different iterators that do not share a common ancestor. STL iterators are thus more performant (facilitate inlining), but they are applicable to a narrower range of problems. In particular, they are not applicable to our problem, which requires dynamic binding as illustrated in Figure 3.

Figures 4–8 include the complete database implementation, and Figure 9 exemplifies how to define one database instance. We shall now address these figures one by one.

Figure 4 shows Sorter_t, which is the abstract "aggregate" interface (for each sorting criterion there will be one Sorter_t). Figure 5 uses Sorter_t to implement the database in a straightforward way. If Sorter_t's insertion, deletion, and lookup are $O(logN)$, and if its begin and end are $O(1)$, then the database meets our complexity requirements (Table 2).

```
struct Sorter_t {

  virtual ~Sorter_t() {}

  virtual void   add  (int i) = 0;
  virtual void   del  (int i) = 0;
  virtual Iter_t find (int i) = 0;
  virtual Iter_t begin()      = 0;
  virtual Iter_t end  ()      = 0;
};
```

Figure 4. *The aggregate (pure virtual interface).*

```
struct Database_t {
  std::vector<Sorter_t*> v;
  const int              K;
  Database_t(const std::vector<Sorter_t*>& u) : v(u), K(u.size()) {  }
  void   add  (int i)        { for(int k=0; k<K; k++) v[k]->add(i);  }
  void   del  (int i)        { for(int k=0; k<K; k++) v[k]->del(i);  }
  Iter_t find (int k, int i) { return v[k]->find(i);                 }
  Iter_t begin(int k)        { return v[k]->begin();                 }
  Iter_t end  (int k)        { return v[k]->end();                   }
};
```

Figure 5. *The database encapsulates a vector of aggregates.*

```
// IB_t  =  Iterator Base Type
// IA_t  =  Iterator Adapter Type


struct IB_t {

  virtual ~IB_t() {}


  virtual bool  operator!=(const IB_t& r)= 0;
  virtual IB_t& operator= (const IB_t& r)= 0;
  virtual IB_t& operator++()             = 0;
  virtual int   operator* ()             = 0;
  virtual IB_t* clone     () const       = 0;
};
```

```
template <typename IntSetIter_t> struct IA_t : public IB_t {

  IntSetIter_t i;

  const IA_t& dc(const IB_t& r) // dc = downcast (IB_t to IA_t)
      { return *dynamic_cast<const IA_t*>(&r); }

  IA_t(IntSetIter_t iter) : i(iter)  {}

  virtual bool  operator!=(const IB_t& r) { return i != dc(r).i;    }
  virtual IB_t& operator= (const IB_t& r) { i=dc(r).i; return *this;}
  virtual IB_t& operator++()              { ++i; return *this;      }
  virtual int   operator* ()              { return *i;              }
  virtual IB_t* clone     () const        { return new IA_t(i);     }
};
```

Figure 6. *Left: the abstract (pure virtual) iterator interface IB_t. Right: a concrete implementation IA_t of the iterator interface. As the latter is generic, it in fact constitutes a family of concrete implementations. Specifically, it adapts any std::set<int,C>::iterator to the IB_t interface, regardless of the specific type of the comparator C.*

```
struct Iter_t {

  IB_t *p;

  Iter_t(const Iter_t& i)             {p=i.p->clone();      }
  Iter_t(const IB_t&   i)             {p=i.clone();         }
  ~Iter_t()                           {delete p; p=0;       }

  bool    operator!=(const Iter_t& r) {return *p != *r.p;   }
  Iter_t& operator++()                {++(*p); return *this;}
  int     operator* ()                {return **p;          }
  Iter_t& operator= (const Iter_t& r)
                    {delete p; p=r.p->clone(); return *this;}

};
```

Figure 7. *The Iter_t proxy rids users from the need to work with pointers to iterators, and from having to explicitly deallocate them. This is the class which is used in Figures 4–5.*

```
template<typename IntSet_t>
struct Container_t : public Sorter_t {

  IntSet_t s;
  typedef typename IntSet_t::iterator INative_t;
  Iter_t wrap(const INative_t& i)
                 {return Iter_t( IA_t<INative_t>(i) );}

  Container_t() {}
  virtual void   add  (int i) {s.insert(i);            }
  virtual void   del  (int i) {s.erase(i);             }
  virtual Iter_t find (int i) {return wrap(s.find(i));}
  virtual Iter_t begin()      {return wrap(s.begin());}
  virtual Iter_t end  ()      {return wrap(s.end()  );}
};
```

Figure 8. *The generic (template) Container_t adapts any std::set<int,C> type to the Sorter_t interface, regardless of the type of C.*

Figure 6 (left) shows IB_t, which stands for “iterator base type”. This is the abstract iterator interface. It declares all the pointer-like iterator operations as pure virtual. For reasons to be shortly addressed, IB_t is not the iterator type used in Figures 4–5. For the same reasons, in addition to the pointer-like operations, IB_t also declares the clone operation, which returns a pointer to a copy of the iterator object; the copy resides on the heap and is allocated with new.

As noted, the *concrete* iterators and containers we use as examples are the highly optimized STL containers and iterators. STL std::sets are suitable for our purposes, as they sort unique items by user-supplied criteria, and they meet our complexity requirements. However, we cannot use STL containers and iterators as is. We must adapt them to our interfaces. Figure 6 (right) shows IA_t, which stands for “iterator adapter type”. This generic class adapts any set<int,C>::iterator type to the IB_t interface, regardless of the actual type of the comparator C. Having to handle different iterator types necessitates IA_t’s genericity.

IB_t and IA_t are seemingly all that we need to complete the implementation of our database. But technically runtime polymorphism only works through pointers or references, typically to heap objects. While in principle we could define Iter_t (from Table 2) to be a pointer, this would place the burden of explicitly delete-ing iterators on the users, which is unacceptable. The solution is to define Iter_t as a proxy to

```
struct lt {
  bool operator() (int x, int y) const {return x < y;}
};
struct gt {
  bool operator() (int x, int y) const {return x > y;}
};

Container_t< std::set<int,lt> > cont_lt;
Container_t< std::set<int,gt> > cont_gt;

std::vector<Sorter_t*> v;
v.push_back( &cont_lt );
v.push_back( &cont_gt );

Database_t db(v);
```

Figure 9. *Creating a database that utilizes two sorting criteria, under the design that abstracts the iterator, which is implemented in Figures 4–8. (Variables with lt or gt types are function objects.)*

```
bool cmp_lt(int x, int y) {return x < y;}
bool cmp_gt(int x, int y) {return x > y;}

typedef bool (*CmpFunc_t) (int x, int y);
typedef std::set<int,CmpFunc_t> Sorter_t;
typedef Sorter_t::iterator Iter_t;

Sorter_t set_lt( cmp_lt );
Sorter_t set_gt( cmp_gt );

std::vector<Sorter_t*> v;
v.push_back( &set_lt );
v.push_back( &set_gt );

Database_t db(v);
```

Figure 10. *Creating the database with the design that abstracts the comparator is simpler, requiring only Figure 5 and three additional type definitions. (Compare with Figure 9.)*

an IB_t pointer, as shown in Figure 7. We can see that Iter_t manages the pointer without any user intervention.

Figure 8 completes the picture by showing Container_t, the generic class that adapts any std::set<int,C> type to the Sorter_t interface, regardless of the type of C. Once again, having to handle different std::set types means Container_t must be generic. Notice how Container_t uses its wrap method to transform the STL iterator into an Iter_t.

Figure 9 demonstrates how the database may be defined. This example uses two sorting criteria in the form of two comparator classes: lt (less than) and gt (greater than), resulting in ascending and descending sequences. Two corresponding std::sets are defined and adapted to the Sorter_t interface by using the generic Container_t. Although the two containers have different types, they have a common ancestor (Sorter_t), which means that they can both reside in the vector v that is passed to the database constructor.

2.2.1 Drawbacks of Using an Abstract Iterator

Our Database_t has some attractive properties. It efficiently supports a simple, yet powerful set of operations (as listed in Table 2), and it is flexible, allowing to easily configure arbitrary collections of sorting criteria. The price is the overhead of abstraction and of runtime polymorphism.

Let us compare the overheads of using a database that has only one sorting criterion (K=1) to using a native std::set directly. Obviously, the two are functionally equivalent, but there are several sources of added overhead.

Firstly, the five set operations listed in Table 2 require an additional virtual function call, as they are invoked through the Sorter_t base class. Conversely, when using std::sets to invoke the same operations, no virtual calls are involved.

Secondly, those operations that return an iterator require dynamic memory allocation through new; this memory is later deleted when the iterators go out of scope. In contrast, std::sets do not invoke new or delete in these operations.

Finally, every iterator operation (increment, dereference, equality test, and assignment) incurs an additional virtual call overhead when (indirectly) used through the IB_t interface. This price might be heavy when compared to native std::set iterators, because the latter are not only non-virtual, but are also inlined. The overhead is magnified by the fact that iterator operations are typically sequentially invoked for N times when traversing the items of the container.

2.3 Using an Abstract Comparator

There is a second standard way to implement our database, which is far simpler. In Section 2.2, we used a collection of std::set<int,C> containers with different C comparators in order to implement the different sorting criteria. This mandated us to deal with the fact that the containers (and associated iterators) have different types. We have done so by abstracting the variance away through the use of the aggregate and iterator interfaces (Sorter_t and IB_t), and by adapting the std::sets and their iterators to these interfaces.

Multiple sorting criteria may be alternatively implemented by abstracting the comparator C, such that all std::sets use the same comparator *type*, but each is associated with a different comparator *instance*. As the sets agree on C, they have identical types, and so their iterators have identical types too. Our implementation would therefore be exempt from handling type variance.

Indeed, to implement this design, we only need the code in Figure 5 along with the following three type definitions:

```
typedef bool (*CmpFunc_t) (int x, int y);
typedef std::set<int,CmpFunc_t> Sorter_t;
typedef Sorter_t::iterator Iter_t;
```

(we no longer need the code in Figures 4, 6, 7, and 8).

A CmpFunc_t variable can hold pointer-to-functions that take two integers as input and return true iff the first is "smaller" than the second. The variable is not bound to a specific value and thus abstracts the comparator away.[3] According-

[3] If we need to add state to the comparison functions and turn them into object functions, we could do so by using a comparator that has a CmpFunc_t

ingly, we define the Sorter_t type as set<int,CmpFunc_t>, which eliminates the need for Figures 4 and 8. We likewise define the iterator Iter_t to be set<int,CmpFunc_t>::iterator, which eliminates the need for Figures 6 and 7.

Figure 10 shows how the new implementation of the database may be instantiated. Similarly to the example given in Figure 9, we use two sorting criteria. But this time, instead of function objects, we use ordinary functions: cmp_lt and cmp_gt; both have a prototype that agrees with the CmpFunc_t type, and so both can be passed to constructors of objects of the type Sorter_t. We next instantiate two objects of the Sorter_t type, set_lt and set_gt, and, during their construction, we provide them with the comparator functions that we have just defined. (Sorters and comparators are associated by their name). As planned, we end up with two objects that have the same *type* but employ different comparator *instances*. We can therefore push the two objects to the vector v, which is passed to the database constructor.

2.3.1 Drawbacks of Using an Abstract Comparator

At first glance, it appears that abstracting the comparator yields a cleaner, shorter, and more elegant implementation than abstracting the iterator (instead of Figures 4–8 we only need Figure 5). Moreover, abstracting the comparator does not generate any of the overheads associated with abstracting the iterator (see Section 2.2.1), because we do not use the abstraction layers depicted in Figure 3. It consequently seems as though abstracting the comparator yields a solution that is superior in every respect. But this is *not* the case. There is a tradeoff involved.

Sorted containers like std::set, which are required to deliver $O(logN)$ performance, are inevitably implemented with balanced trees. When a new item is inserted to such a tree, it is compared against each item along the relevant tree path. If the comparator is abstracted, each comparison translates to a non-inlined function call (this is the price of runtime polymorphism). But if the comparator is not abstracted, its code is typically inlined, as it is known at compile time. For example, in Figure 9, comparisons resolve into a handful of inlined machine operations. This observation applies to insertion, deletion, and lookup. (Later, we quantify the penalty of abstract comparators and show it is significant.)

We conclude that there are no clear winners. If users want to optimize for iteration, they should abstract the comparator. (Comparators do not affect the iteration mechanism in any way, as discussed in the next section.) But if they want to optimize for insertion and deletion, they should abstract the iterator instead. In the next section, we show that it is in fact possible to obtain the benefits of both approaches.

3. Independent Iterator: The New Approach

Let us reconsider the two alternative database designs from the previous section. The specification of the problem (Table 2) requires supporting iteration over the database according to multiple sorting criteria using the same iterator type. We have utilized std::sets with different comparators to allow for the different sorting criteria, and we were therefore required to face the problem of having multiple iterator types instead of one.

The heart of the problem can be highlighted as follows. Given two comparator types C1 and C2, and given the following type definitions

```
typedef std::set<int,C1> S1_t; // sorting criterion #1
typedef std::set<int,C2> S2_t; // sorting criterion #2
typedef S1_t::iterator I1_t;
typedef S2_t::iterator I2_t;
```

the iterator types I1_t and I2_t are different. In the previous section we have dealt with this difficulty by either

1. adapting I1_t and I2_t to an external iterator hierarchy rooted by a common ancestor which is an abstract iterator (Section 2.2), or by
2. morphing I1_t and I2_t into being the same type, by favoring to use *multiple instances* of one abstract comparator type, over using *multiple types* of comparators that are unrelated (Section 2.3).

Both solutions required trading off some form of compile-time polymorphism and excluded the corresponding inlining opportunities. Importantly, the need for abstraction has arisen due to a perception that has, so far, been undisputed: that if we instantiate a generic class (std::set) with different type parameters (C1 and C2), then the type of the corresponding inner classes (I1_t and I2_t) will differ. We challenge this perception, both conceptually and technically.

3.1 The Conceptual Aspect

As noted, the data structure underling std::sets is inevitably a balanced search tree, because of the $O(logN)$ STL-mandated complexity requirement. A distinct feature of search trees is that the order of the items within them is *exclusively* dictated by the structure of the tree [12]. Specifically, by definition, the minimal item in a tree is the leftmost node; and (assuming the search tree is binary) the successor of each node x is the leftmost item in the subtree rooted by $x.right$ (if exists), or the parent of the closest ancestor of x that is a left child. These two algorithms ("minimal" and "successor") completely determine the traversal order. And both of them *never* consult the keys that reside within the nodes. Namely, the algorithms are entirely structure-based.

As keys are not consulted, then, obviously, the comparator function associated with the tree (which operates on keys) is unneeded for realizing an in-order traversal. Likewise, as nodes are not created or destroyed within the two algorithms, the memory allocator of the tree is unneeded too.

data member, which is invoked in the "operator()" of the class [29].

```
template<typename T, typename C, typename A> class set {

  public:
    class iterator {
        // code does not utilize C or A ...
    };

    // ...
};
```

```
template<typename T> class iterator {
    // ...
};

template<typename T, typename C, typename A> class set {
  public:
    typedef iterator<T> iterator;
    // ...
};
```

Figure 11. *Left: the iterator is dependent on the comparator C and the allocator A. Right: the iterator is independent.*

It follows that, by definition, in-order traversal is an activity which is independent of comparators and allocators. And since iterators are the technical means to conduct such a traversal, then, conceptually, iterators should be independent of comparators and allocators too. In particular, there is no conceptual reason that requires std::sets that disagree on comparators or allocators to have different iterator types.[4]

3.2 The Technical Aspect

The question is therefore whether we are able, technically, to eliminate the unwarranted dependencies and utilize a single iterator type for different integer std::sets that have different comparators or allocators. The answer is that we can as shown in Figure 11. All that is required is removing the code of the iterator class from within the internal scope of the set, placing it in an external scope, and preceding it with a template declaration that accurately reflects the dependencies, including only the item type T (integer in our example) and excluding the comparator and allocator types C and A. The removed iterator code is then replaced with an alias (typedef) that points to the iterator definition that is now external. The functionality of the alias is identical to that of the original class for all practical purposes.[5]

We conclude that our goal is achievable. Namely, it is possible to define a nested class of a generic class such that the nested class only depends on some, but not all, of the generic parameters. Thus, there is no need to modify the language or the compiler. Rather, the issue is reduced to a mere technicality: how the generic class is implemented, or, in our case, how the STL is implemented.

Table 1 lists several mainstream compilers and specifies if the std::set iterator class that they make available (in their default mode) is dependent on the comparator or allocator. It should now be clear that this specification is a product of the STL that is shipped with the compiler.

Table 3 lists the four most widely used STL implementations. All the compilers in Table 1 that are associated with a dependent iterator make use of Dinkum STL; the exception is the compiler by Sun, which uses an implementation that is based on an early commercial version of RogueWave STL. Conversely, the compilers with an independent iterator all make use of the GNU open source libstdc++ STL.

STL	iterator
Dinkum	dependent
libstdc++	independent
STLPort	independent
RogueWave	both (depends on version and mode)

Table 3. *Standard template library implementations.*

Some compilers ship with more than one STL implementation and allow users, through compilation flags, to specify whether they want to use an alternative STL. For example, when supplied with the flag “ -library=stlport4”, the Sun compiler will switch from its commercial RogueWave-based implementation to STLport; the iterator will then become independent of the comparator and allocator.

Interestingly, the iterator of the most recent RogueWave (open source) is dependent on or independent of the comparator and allocator, based on whether the compilation is in debug or production mode, respectively. The reason is that, in debug mode, one of the generic parameters of the iterator is the specific std::set type with which it is associated (which, in turn, depends on the comparator and allocator). The debug-iterator holds a pointer to the associated std::set instance and performs various sanity checks using the begin and end methods (e.g., when the iterator is dereferenced, it checks that it does not point to the end of the sequence). Such sanity checks are legitimate and can help during the development process. But there is no need to make the iterator dependent on its std::set in order to perform these checks; this is just another example of an unwarranted dependency that delivers no real benefit. Indeed, instead of the std::set, the iterator can point to the root node of the balanced tree (which, as explained in Section 3.1, should not depend on the comparator and allocator); the begin and end of the tree are immediately accessible through this root.

3.3 The Database with an Independent Iterator

To implement our database with the new approach we need Figures 4, 5, and 8, as well as the following type definition

```
typedef std::set<int, SomeC, SomeA>::iterator Iter_t;
```

It does not matter which C or A we use, because we assume that the iterator do not depend on them. Figure 9 ex-

[4] Technically, to share an iterator type, std::sets must agree on the following nested types and nothing else: value_type (T), pointer (to T), const_pointer (to T), and difference_type (of subtracting two pointers).

[5] Alternatively, we could have (1) defined a base class for std::set that only depends on T and (2) cut-and-pasted the iterator to the base class's scope.

emplifies how to instantiate this type of database. This is the same example we used in Section 2.2 ("abstract iterator"). But, in contrast to Section 2.2, we now do not need Figures 6–7 (the external iterator hierarchy), because all set<int,C,A>::iterator types are one and the same regardless of C or A, and so there is no reason to introduce an abstraction layer to hide the differences.

Importantly, notice that, with the current type definition of Iter_t, we now use SCARY assignments in all the figures involved (4, 5, and 8). Specifically, every return statement in every method within these figures that has an Iter_t return-type is such a statement, because the returned value is associated with containers that utilize different comparator types. Only if these containers share the same iterator type will this code compile. Thus, this implementation is only valid with STLs like libstdc++, which define an independent iterator; it will *not* compile with STLs like Dinkum.

3.4 Advantages of Using an Independent Iterator

The overheads induced by the new approach are similar to that of the abstract iterator design (Section 2.2.1) in that we cannot avoid using the Sorter_t interface. This is true because we are utilizing different types of std::sets (have different comparator types), and so the std::sets must be adapted to conform to one interface in order to facilitate uniform access (which is required by the database implementation in Figure 5). Every operation that is done through the Sorter_t interface involves an added virtual function call, which is entirely avoided when utilizing the abstract comparator design. And since there are K sorting criteria, there are actually K such extra function invocations.

This, however, does not mean that the new approach is inferior. In fact, the opposite is true. To understand why, assume that the database is currently empty, and that we have now added the first item. In this case, contrary to our claim, the abstract comparator design is superior, because, as noted, the new approach induces K extra virtual function calls that are absent from the abstract comparator design.

We now add the second item. While the abstract comparator design avoids the virtual calls, it must compare the second item to the first. This is done with the help of K pointers to comparison functions and therefore induces the overhead of K function invocations. Conversely, the comparisons performed by the std::sets of the new approach are inlined, because the implementation of the comparator types is known at compile time. Thus, for the second item, the two designs are tied: K vs. K invocations.

We now add the third element. With the new approach, there are still only K function calls; nothing has changed in this respect. But with the abstract comparator design, there might be up to $2K$ function invocations (and no less than K), depending on the values of the items involved.

In the general case, whenever a new item is added, the abstract comparator design performs $O(K \cdot logN)$ function invocations ($logN$ comparisons along each of the K insertion paths), whereas the new approach performs exactly K. The same observation holds for deletion and lookup.

Focusing on iteration, we note that the new approach does not (de)allocate iterators through new and delete. The abstract comparator design still has the advantage that its begin and end are not virtual. But in accordance to the iteration procedure shown in Figure 2, this advantage occurs only once per iteration, during which N elements are traversed. In both designs, the pointer-like iterator operations that are exercised N times are identical, as both directly utilize the native set<int>::iterator without abstraction layers. Thus, the advantage due to the one extra virtual call quickly becomes negligible as N increases. We later show that the difference is noticeable only while $N \leq 4$.

We conclude that, excluding a few small N values, the new approach is superior to the two standard designs: It is better than the abstract iterator design when iterating and finding, and it is better than the abstract comparator design when finding, adding, and deleting.

3.5 Consequences

In relation to our running example, we contend that the independence of the iterator should be made part of the STL specification, or else programmers would be unable to use the new approach if their environment does not support the right kind of STL, or if they wish to write portable programs that compile on more than one platform.

But this is just one example. The more general principle we advocate is that, when designing a generic class, designers should (1) attempt to minimize the dependencies between the class's type parameters and nested types, and (2) should make the remaining dependencies part of the user contract, declaring that no other dependencies exist.

Reducing dependencies directly translates to increased compile-time interchangeability; and explicitly declaring that no other dependencies exist makes it possible for programmers to leverage this increased interchangeability for writing faster programs.

3.6 Disadvantages of Using an Independent Iterator

Independent iterators make one problem slightly worse. Assume, e.g., that vectors v1 and v2 hold elements of type T but utilize different allocator types. The following error

```
p = v1.begin();
q = v2.end();
std::sort(p,q); // error!
```

can be caught at compile time if v1 and v2 have different iterator types (which is the case if the iterator depends on the allocator); otherwise, the error can only be caught at runtime. Such problems do occur in real life, however, the only complete solution is to have related iterators extracted from their container by code rather than by hand, as is made possible by C++0x, the upcoming revision of ISO C++.

4. Experimental Results: Runtime

In this section we evaluate the two standard solutions described in Section 2 against our proposal from the previous section. We denote the three competing database designs as:

1. the "iterator design" (Section 2.2),
2. the "comparator design" (Section 2.3), and
3. the "new design" (Section 3.3).

We conduct a two-phase evaluation. In Section 4.1, we use microbenchmarks to characterize the performance of each individual database operation. And in Section 4.2, we evaluate the overall effect on a real application.

The experiments were conducted on a 2.4 GHz Intel Core 2 Duo machine equipped with 4GB memory and running lenny/sid Debian (Linux 2.6.20 kernel). The benchmarks were compiled with GCC 4.3.2, using the "-O2" flag. While running, the benchmarks were pinned to a single core, and times were measured using the core's cycle counter; the reported results are averages over multiple runs. Except from the default Debian daemons, no other processes were present in the system while the measurements took place.

4.1 Microbenchmarks

We use four microbenchmarks to measure the duration of adding, deleting, finding, and iterating through the items. Figure 12 displays the results. Durations are presented as a function of N (number of database items), and N is shown along the x axis. The "add" microbenchmark sequentially adds N different items to an empty database, where N is 2^i for $i = 0, 1, 2, ..., 22$. The y axis shows how long it took to perform this work, normalized (divided) by $N{\cdot}K$. (K was chosen to be 2, as shown in Figures 9–10.) The y axis thus reflects the average time it takes to add one item to one container associated with one sorting criterion.

The other three microbenchmarks are similarly defined and normalized: "delete" sequentially erases the N items in the order by which they were inserted; "find" looks up each of the N items within the database according to each of the K sorting criteria (and checks that the returned iterator is different than the corresponding end of sequence); and "iterate" traverses the N items (using the procedure shown in Figure 2) according to each of the K sorting criteria.

The results coincide with our analysis from Section 3.4.

Comparator vs. new Figure 12 illustrates that the new design adds, deletes, and finds items faster than the comparator design. Indeed, these activities require repeated item comparisons along the associated search tree paths; the comparisons translate to function invocations in the comparator design, but resolve into inlined code in the new design. Iteration, on the other hand, involves no comparisons, and so the performance of the comparator and new designs is similar.

Figure 13(a) shows the corresponding relative speedup, defined as the ratio of the duration it takes to perform each operation under the two competing designs. (Values bigger than 1 indicate the new design is faster.) Initially, for small N values, the comparator design may be faster. This happens because the new design utilizes the Sorter_t interface and thus induces one extra virtual function call (two in the case of the "iterate" benchmark: begin and end). But when N is increased, the relative weight of this overhead decreases, as more and more items must be compared ("iterate": must be traversed), such that beyond N=4 the speedup is always bigger than 1 ("iterate": equal to 1).

We were initially surprised by the fact that the "find" speedup is smaller than that of "add" and (sometimes) of "delete". As the latter perform a lot more work that does not involve comparisons (allocation, deallocation, and balancing), we anticipated that the relative weight of the comparisons would be smaller. It turns out that "add" and "delete" actually require more comparisons, because the underlying (red black) search tree is generally "less balanced" while they execute. The reason is that, when we repeatedly add items and monotonically grow the tree, we systematically encounter those cases that trigger the balancing activity, which occurs only when the tree is not "balanced enough". (Monotonically deleting items has the same affect.) Such cases always involve an extra comparison, and "find" never encounters these cases because it does not alter the tree.

Overall, the speedup behavior is the following. It goes up (for the reasons discussed above), reaches a kind of steady state that peaks at nearly 1.7, and then "falls off a cliff" to a level of around 1.15. We investigated the reason that causes the fall and discovered that it is tightly connected to the size of the L2 cache. Figure 14 plots the "delete" speedup curve and superimposes on it the associated resident set size (RSS) as reported by the operating system through the proc filesystem [44]; the RSS reflects the size of physical memory the benchmark utilized. On our testbed machine, the size of the L2 cache is 4MB, and according to Figure 14, the biggest database size to fit within the L2 is N=64K. We can indeed see that immediately after that N, the speedup drops. The reason is that memory accesses can no longer be served by the cache and require going to main memory. As such accesses may take hundreds of cycles, the relative benefit of inlined comparisons within the new design diminishes.

Iterator vs. new By Figure 12, the time to add and delete items by both designs is similar, which should come as no surprise because they utilize the same exact code to perform these activities. The new design, however, finds items and iterates through them faster than the iterator design. The reason is that, with the iterator design, both activities dynamically (de)allocate iterator instances through new and delete; moreover, every operation applied to these instances is realized through an abstract interface and induces a virtual function call (as opposed to the new design that inlines these operations). This was explained in detail in Section 3.4.

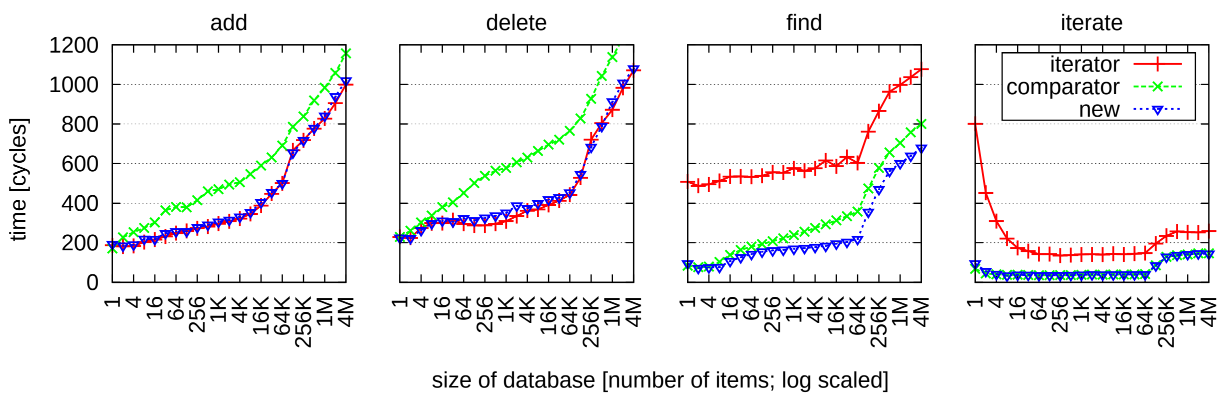

Figure 12. *The results of the four microbenchmarks as achieved by the three competing database designs.*

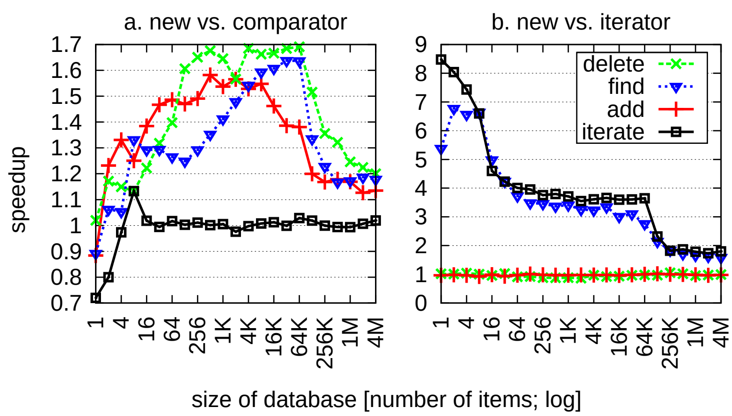

Figure 13. *The microbenchmark speedups achieved by the new design relative to the comparator (a) and iterator (b) designs.*

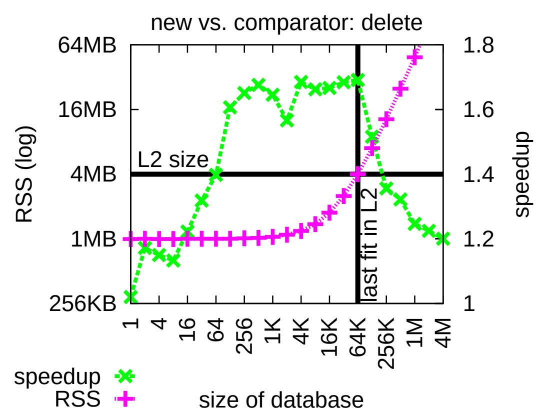

Figure 14. *Speedup drops when the microbenchmark's resident set size (RSS) no longer fits in the L2 cache.*

The associated speedup, shown in Figure 13(b), can therefore be explained as follows. Initially, for small N values, the dynamic (de)allocation is the dominant part, as there are relatively few items to process. But as N increases, the price of dynamic (de)allocation is amortized across more items, causing the speedup ratio to get smaller. The speedup then enters a steady state, until N=64K is reached and the database no longer fits in L2, at which point it drops to a level of around 1.75.

Throughout the entire N range, the "iterate" speedup is higher than that of "find", because the former involves an additional virtual call ("find" only compares the returned iterator to end, whereas "iterate" also increments the iterator).

4.2 Real Application

To evaluate the new design in the context of a real application, we use an in-house scheduler simulator, which is used for researching and designing the scheduling subsystem of supercomputers such as the IBM BlueGene machines. The simulator is capable of simulating the schedulers of most machines within the top-500 list [15], and it has been extensively used for research purposes [41, 21, 20, 50, 46]. Others have implemented similar simulators [40, 16, 31].

The workload of supercomputers typically consists of a sequence of jobs submitted for batch execution. Accordingly, years-worth of logs that record such activity in real supercomputer installations are used to drive the simulations. The logs are converted to a standard format [10] and are made available through various archives [42, 43]. Each log includes a description of the corresponding machine and the sequence of submitted jobs; each job is characterized by attributes such as its arrival time, runtime, and the number of processors it used. The simulator reads the log, simulates the activity under the design that is being evaluated, and outputs various performance metrics. For the purpose of performance evaluation, each log is considered a benchmark.

The simulator is a discrete event-driven program. Events can be, e.g., job arrivals and terminations. Upon an event, the scheduler utilizes two main data structures: the wait queue and the runlist. It inserts arriving jobs to the wait queue and removes terminating jobs from the runlist. It then scans the runlist to predict resources availability, and it scans the wait queue to find jobs that can make use of these resources. According to various dynamic considerations, the order of the job-scanning may change; the algorithm that makes use

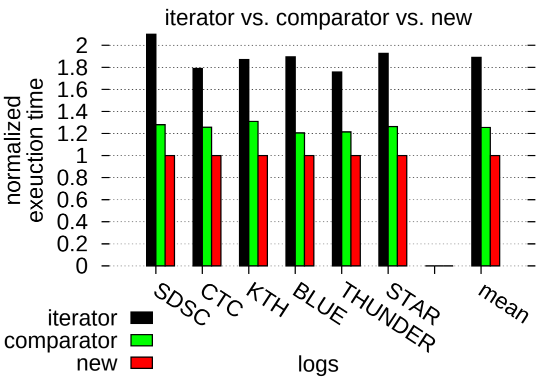

Figure 15. *Normalized execution time of the three simulator versions, when simulating six activity logs.*

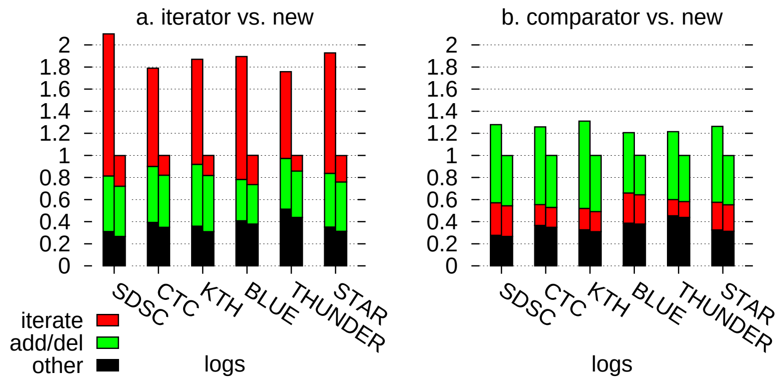

Figure 16. *Breaking the execution time from Figure 15 to three disjoint components: addition/deletion of items, traversal through the items, and the rest.*

of the scanning is otherwise the same. Adequate candidates are removed from the wait queue and inserted to the runlist.

It follows that the main data structures of the simulator must support functionality similar to that of the database we have developed earlier. Originally, the simulator was implemented using the classic iterator design pattern. We have modified the simulator and implemented the other two competing designs, such that the one being used is chosen through a compilation flag. The evaluated scheduler required four sorting criteria for the wait queue (job arrival time, runtime, user estimated runtime, and system predicted runtime) and three for the runlist (termination time based on: real runtime, user estimated runtime, and system predicted runtime). The data structures store job IDs (integers) that serve as indices to a vector that is initialized at start-up and holds all the jobs (and, thus, comparators refer to this vector).

Figure 15 shows the time it takes to complete the simulation when simulating six workload logs.[6] All execution times are normalized by that of the new design, on a per-log basis. We can see that the execution time of the iterator design is x1.7 to x2.1 slower than that of the new design, and that the execution time of the comparator design is x1.2 to x1.3 slower, depending on the log.

Figure 16 breaks the execution time to three disjoint components: cycles that were spent on adding or deleting jobs to/from the wait queue and runlist, cycles that were spent on traversing the jobs, and all the rest. (The simulator does not utilize the find operation.) We have shown in Section 4.1 that the new design is superior to the comparator design in terms of adding and deleting; and indeed, the majority of the difference in their executions times is caused by addition and deletion. Likewise, we have shown that the new design is superior to the iterator design in terms of traversing; and indeed, the majority of the difference in executions times between these two designs is caused by iteration.

[6] The logs span months to years and contain tens to hundreds of thousands of jobs submitted by hundreds of users operating within different sites under different load conditions; further details can be found in the Parallel Workload Archive site [42].

A minor part of the difference is caused by the other operations, which are identical across all designs. We speculate that this is caused by caching effects that are triggered by the less efficient parts of the program.

5. Techniques to Reduce Code Bloat

Compilers generate object code. In this section, we focus on how the code's size (in bytes) can be affected by reducing unneeded dependencies between the members and type parameters of a generic class. To this end, we continue to use STL containers and iterators. But the discussion no longer revolves around the three designs from the previous section. Rather, it focuses on the impact of using multiple type parameters to instantiate a single class, e.g., as is done in Figure 9, where we use *two* comparator types (lt and gt) as type parameters. In Figure 10, we only use *one* type parameter (CmpFunc_t) and so our discussion does not apply. The more type parameters that are used, the bigger the object code that is emitted. This increase in code is sometimes referred to as *bloat*, and this section is about reducing it.

5.1 What We Have Already Achieved

Let us reconsider Figure 11. On its left, the iterator is inner and thus depends on the comparator and allocator. The design on its right defines the iterator outside and removes the unneeded dependencies. We denote these two designs as "inner" and "outer", respectively. In Section 4, we have shown how to leverage the outer design to write faster programs. Here, we additionally argue that it also allows for reducing the bloat. To see why, consider the following snippet that copies two integer std::sets into two matching arrays.

```
std::set<int,lt> u; // assume u holds N elements
std::set<int,gt> v; // v holds N elements too

int arr1[N];
int arr2[N];

std::copy( u.begin(), u.end(), arr1 ); // copy u to arr1
std::copy( v.begin(), v.end(), arr2 ); // copy v to arr2
```

Suppose we (1) compile this snippet with an STL that utilizes the inner design, (2) generate an executable called a.exe, and (3) run the following shell command, which prints how many times the symbol std::copy is found in a.exe:

```
nm –demangle a.exe | grep -c std::copy
```

The result would be 2, indicating that the function std::copy was instantiated twice. In contrast, if we use an STL that utilizes the outer design, the result would be 1, reflecting the fact that there is only one instantiation. The reason for this difference is that, like many other standard C++ algorithms, std::copy is parameterized by the iterators' type:

```
template<typename Src_Iter, typename Dst_Iter>
Dst_Iter std::copy(Src_Iter begin, Src_Iter end, Dst_Iter target);
```

With the inner design, the iterator types of u and v are different due to their different comparators, which means there are two Src_Iter types, resulting in two instantiations of copy. The outer design has an independent iterator, yielding only one Src_Iter and, hence, only one instantiation. The same argument holds when using several allocator types.

We conclude that, when unneeded dependencies exist, every additional type parameter associated with these dependencies results in another instantiation of the algorithm. This type of bloat is unnecessary and can be avoided by following the principle we advocate and eliminating the said dependencies. Thus, in addition to allowing for faster code, our proposal also allows for code that is more compact.

5.2 What We Can Achieve Further

Our above understandings regarding bloat and how to reduce it can be generalized to have a wider applicability as follows.

The outer design is successful in reducing the bloat of standard generic algorithms like std::copy, because such algorithms suffer from no unneeded dependencies. This is true because (1) every type parameter that is explicitly associated with any such algorithm is a result of careful consideration and unavoidable necessity; and because (2) such algorithms are global routines that reside in no class and hence are not subject to implicit dependencies.

The latter statement does not apply to algorithms that are methods of a generic class. For example, all the member methods of std::set<T,C,A> implicitly depend on the key type T, the comparator type C, and the allocator type A. We observe that this triple dependency occurs even if, logically, it should not. And we note that this is exactly the same observation we have made regarding member classes (iterators). We contend that this observation presents a similar opportunity to reduce the bloat.

5.3 Hoisting

Others have already taken the first step to exploit this opportunity, targeting the case where a method of a generic class is logically independent of all the generic parameters. For example, the size method that returns the number of elements stored by the std::set[7]

```
template<typename T, typename C, typename A>
size_type set<T,C,A>::size() const { return this->count; }
```

This method just returns an integer data member (the alias size_type is some integer type) and so its implementation is independent of T, C, and A. Yet, for every T/C/A combination, the compiler emits another identical instantiation.

The well-known and widely used solution to this problem is *template hoisting* [8]. The generic class is split into a non-generic base class and a generic derived class, such that all the members that do not depend on any of the type parameters are moved, "hoisted", to the base class. In our example, these members are size and count, and their hoisting ensures that size is instantiated only once. Importantly, hoisting induces no performance penalty, as none of the methods are made virtual and no runtime polymorphism is involved.

Most STL implementations use hoisting to implement the standard associative containers. These are set, multiset, map, and multimap. (Sets hold keys, maps hold key/data pairs, and multi-containers can hold non-unique keys.) All the libraries listed in Table 3 implement these containers using one generic red-black tree class. Henceforth, we only consider the latter. As explained in Section 3.1, iteration-related code and the balancing code of the tree need not depend on T, C, and A, because they are strictly based on the structure of the tree. And indeed, these routines are typically hoisted and operate on pointers to the non-generic base class of the tree's node. Thus, there is only one instantiation of the tree "rebalance" method for all the associative containers.

5.4 Generalized Hoisting

We contend that hoisting can be generalized to reduce the bloat more effectively. Our claim is motivated by the following analysis. We have examined the code of the generic red-black tree class of GCC and found that nearly all of its methods either: (1) exclusively depend on T, T/C, or T/A; or (2) can be be trivially modified to have this type of dependency. We therefore propose to decompose the tree in a way that removes the other dependencies.

Figure 17 roughly illustrates this idea. On the left, the red-black tree is defined using one class, so when, e.g., the balancing code is required, every change in T, C, or A will result in another duplicate instantiation of rebalance. The middle of the figure rectifies this deficiency by creating a non-generic base class and by hoisting rebalance. There will now be just one such instance, regardless of how many T/C/A combinations there are. The right of the figure takes the next step and eliminates the remaining unneeded dependencies.

[7] In practice, size is inlined; we assume it is not to allow for a short example.

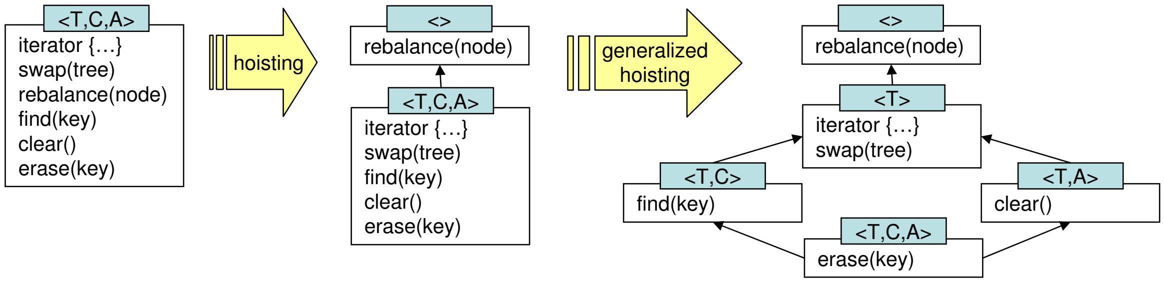

Figure 17. *Generalized hoisting decomposes the generic class to reduce dependencies between members and type parameters. In practice, to avoid the indirection layers caused by a diamond-shape inheritance relation, we will not use multiple inheritance; see details in Section 6.1.*

The erase routine needs the comparator to find an item, and it needs the allocator to deallocate it, so it depends on T/C/A. This is not the case for the find routine, which only depends on T/C, as it merely compares items. The clear routine systematically deallocates the entire tree and does not care about the order dictated by the comparator; it thus depends on only T/A. Finally, the majority of the code of the swap routine (which swaps the content of two trees in a way that does not involve actual copying of items) depends on only T. (The reminder of swap's code will be shortly discussed.) Like swap, as we have discussed in much detail, the nested iterator class only depends on T. In Figure 11 we have suggested to move its definition to an external scope to eliminate its dependency on C/A. Generalized hoisting is an alternative way to achieve this goal.

6. Experimental Results: Code Bloat

We have refactored Rb_tree, the red-black tree underlying all associative containers of GCC's STL, according to the generalized hoisting design principle. This section describes our experience and evaluates its success. We note that we have intentionally constrained ourselves to only applying trivial changes to Rb_tree, even though, in some cases, a more intrusive change would have been more effective.

6.1 Applying Generalized Hoisting to the STL of GCC

Bloat reduction is only applicable to methods that are (1) not inlined or (2) inlined, but invoke methods that are not inlined (directly or indirectly). Henceforth, we refer to such methods as *noninlined*. The code of the remaining methods (that are *not* noninlined) is re-emitted for each invocation; this is inherent to inlining and is unrelated to generics' bloat.

All of GCC's STL associative containers are inlined wrappers of Rb_tree, and their methods translate to invocations of Rb_tree's public methods; this wrapping code leaves no trace after inlining, and is optimized out in its entirety.

Out of the 46 public methods of Rb_tree, 29 are noninlined as listed in Table 4 (we are currently only interested in the method-name column; the other columns are addressed in Section 6.2). We have refactored Rb_tree using three additional classes: Rb_base (depends on only T), Rb_alloc (T/A), and Rb_cmp (T/C), such that Rb_alloc derives std::allocator by default (or the user's allocator if specified), Rb_cmp derives Rb_base, and Rb_tree derives Rb_cmp and has an instance of Rb_alloc. The design is shown in Figure 18. We

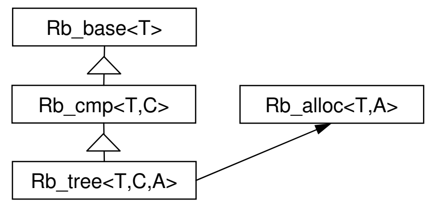

Figure 18. *Our refactored tree does not use multiple inheritance.*

avoid the diamond-shape inheritance depicted in Figure 17 (virtual multiple inheritance), because it complicates the memory layout by introducing certain indirection layers that might incur a performance penalty. (We did not investigate what this penalty is.) Furthermore, beyond what is already in the original Rb_tree, we categorically did not add calls to functions that are not completely inlined so as not to degrade the performance. Namely, we did not add any indirection.

Group *i* in Table 4 includes the methods that reside in Rb_base (swap) or in an external scope (comparison operators). The former is comprised of 40 lines of code, only 2 of which (swaping the comparator and allocator) depend on C/A; we hoist the first 38 lines, and replace the original Rb_tree::swap with an inlined version that (1) calls the hoisted version and (2) invokes the remaining 2 lines.

The comparison operators are inlined calls to the global std::lexicographical_compare, which, like std::copy, operate on our hoisted iterator, and so it depends on only T.

Group *ii* includes the noninlined methods that copy or destroy the tree, or destroy a range of iterators. None of these activities require the comparator, and so this functionality is moved to Rb_alloc. Remaining in Rb_tree is easily splittable code like copying of the comparator (as in swap).

Group *iii* includes the only two routines that actually use noninlined methods from both Rb_cmp and Rb_alloc. These routines need to find a range of iterators associated with a key (there can be more than one in multi-containers) and use Rb_cmp::equal_range for this purpose. Once found, the range is deleted with the Rb_alloc::erase from Group *ii*. No refactoring was needed in this case.

#	method name	original b_0	original d_0	refactored b_1	refactored c_1	refactored a_1	refactored d_1	goodness of fit (R^2) original	goodness of fit (R^2) refactored	diff.
i. Noninlined code from only Rb_base, or external										
1	swap	1	461	369	0	0	45	0.999998	0.999745	48
2	operator >=	104	589	595	0	0	67	0.999998	0.999896	30
3	operator >	104	589	595	0	0	67	0.999998	0.999896	30
4	operator <=	104	589	595	0	0	67	0.999998	0.999896	30
5	operator <	104	589	595	0	0	67	0.999998	0.999896	30
ii. Noninlined code not from Rb_cmp										
6	erase(iterator,iterator)	381	1004	382	0	944	79	1.000000	0.999992	-20
7	destructor	238	459	237	0	403	45	0.999998	0.999966	12
8	clear	236	542	236	0	402	128	0.999998	0.999987	12
9	operator =	471	1680	491	0	1114	534	1.000000	0.999997	11
10	copy constructor	87	1760	179	0	1019	643	0.995402	0.972517	4
iii. Noninlined code from both Rb_cmp and Rb_alloc										
11	erase(key)	375	2080	386	530	942	583	1.000000	0.999999	14
12	erase(key*,key*)	376	2449	392	527	939	952	0.999999	0.999994	14
iv. Noninlined code not from Rb_alloc										
13	insert_equal(iterator,value)	187	1633	208	1556	0	88	0.999996	0.999999	-32
14	insert_equal(value)	124	992	144	928	0	70	0.999990	0.999991	-25
15	insert_equal(iterator,iterator)	177	1493	207	928	0	568	0.999999	0.999999	-32
16	insert_unique(value)	166	1144	212	496	0	580	0.999988	0.999999	21
17	insert_unique(iterator,iterator)	239	1641	272	496	0	1060	0.999999	0.999999	51
18	insert_unique(iterator,value)	188	1893	213	496	0	1363	0.999997	1.000000	8
v. Likewise + entirely contained in Rb_cmp										
19	count(key) const	0	1092	0	1010	0	42	0.999999	0.999980	39
20	count(key)	0	1092	0	1010	0	42	0.999999	0.999979	39
21	rb_verify() const	0	681	0	667	0	28	0.999998	0.999916	-14
22	upper_bound(key) const	0	343	0	269	0	50	0.999915	0.999946	23
23	upper_bound(key)	0	341	0	268	0	50	0.999918	0.999942	23
24	lower_bound(key) const	0	343	0	269	0	50	0.999915	0.999946	23
25	lower_bound(key)	0	341	0	268	0	50	0.999918	0.999942	23
26	find(key) const	0	343	0	269	0	50	0.999915	0.999946	23
27	find(key)	0	341	0	268	0	50	0.999912	0.999939	22
28	equal_range(key) const	0	699	0	508	0	131	0.999970	0.999637	60
29	equal_range(key)	0	695	0	504	0	131	0.999970	0.999634	60

Table 4. *The noninlined methods of GCC's std::Rb_tree. We model the associated object code size (in bytes) with $\bar{s_0}(x,y) = b_0 + d_0xy$ (size of original tree) and $\bar{s_1}(x,y) = b_1 + c_1x + a_1y + d_1xy$ (size of refactored tree). Fitting against the real data is done with the nonlinear least-squares Marquardt-Levenberg algorithm; the resulting R^2 values are nearly 1, indicating the fits are accurate.*

The insertion functions in Group *iv* do not require any Rb_alloc code except from allocating a new node, which is done with a short pure inlined Rb_alloc function. The first insert_equal methods (13–14) are multi-container versions that add the new item even if it is already found in the tree. We move these to Rb_cmp and change them such that instead of getting the new key as a parameter, they get an already allocated node holding that key; we make the new Rb_tree versions inlined calls to the associated Rb_cmp versions, and we incorporate the node allocation as part of the call. These were one-line changes. Method 15 repeatedly invokes method 14 and so remains unchanged.

The refactoring of the insert_unique methods (16–18) was different because they correspond to unique containers (that allocate a new node only if the respective key is not already found in the tree), and they therefore involve a more complex allocation logic. We initially left these methods nearly unchanged, but later realized that they included several calls to an internal function that we wrapped in inlined code, and this repeated code contributed to the bloat. Fortunately, an easy fix was possible. The methods consist of a sequence of conditional branches, such that each branch ends with a call to the internal function; we replace all these calls with a single call positioned after the branches.

The remaining methods, in Group *v*, are query routines that only use the comparator and perform no (de)allocation. We move them in their entirety to Rb_cmp.

6.2 Evaluation

To evaluate the effect of our design, we (1) fix T, (2) systematically vary C and A, and (3) measure the size of the resulting object code on a per-method basis. Let T be an integer type, and let $\{C_1, C_2, ..., C_n\}$ and $\{A_1, A_2, ..., A_n\}$ be n different integer comparators and allocators, respectively. Given a noninlined Rb_tree function f, let f_i^j be one invocation of Rb_tree<T,C$_i$,A$_j$>::f (i.e., the instantiation

of Rb_tree's f when using key type T, comparator type C_i, and allocator type A_j). Given $x, y \in \{1, 2, ..., n\}$, we define $s(x, y)$ to be the size, in bytes, of the file that is comprised of the invocations f_i^j for $i = 1, 2, ..., x$ and $j = 1, 2, ..., y$. For example, $s(1, 1)$ is the size of the object file that only contains the call to f_1^1; $s(1, 2)$ contains two calls: f_1^1 and f_1^2; and $s(2, 2)$ contains f_1^1, f_1^2, f_2^1, and f_2^2.

Figure 19 (left) shows $s_0(x, y)$ (size of original swap) and $s_1(x, y)$ (size of refactored swap), in kilobytes, as a function of the number of comparators $x = 1, ..., 5$ and allocators $y = 1, ..., 5$ (a total of $5 \times 5 = 25$ object files). Most of swap (refactored version) resides in Rb_base, and this part is instantiated only once. In contrast with the original version, the code is re-instantiated for every additional comparator or allocator, which explains why $s_0(x, y)$ becomes bigger than $s_1(x, y)$ at approximately the same rate along both axes.

We hypothesize that the size $s_1(x, y)$ of each noninlined *refactored* method f can be modeled as follows:

$$s_1(x,y) \;\approx\; \bar{s_1}(x,y) \;=\; b_1 + c_1 x + a_1 y + d_1 xy$$

where b_1 is the size of f's code (in bytes) that depends on only T (or nothing) and is thus instantiated only once; c_1 is the size of f's Rb_cmp code (depends on only C and re-emitted for each additional comparator); a_1 is the size of f's Rb_alloc code (depends on only A and re-emitted for each additional allocator); and d_1 is the size of f's Rb_tree code (depends on both C and A and re-emitted for each additional comparator or allocator). We likewise hypothesize that the size $s_0(x, y)$ of each noninlined *original* method can be modeled as

$$s_0(x,y) \;\approx\; \bar{s_0}(x,y) \;=\; b_0 + d_0 xy$$

($c_0 = a_0 = 0$, as the original Rb_tree aggregates all the code and so none of the code is solely dependent on C or A.)

If the models $\bar{s_0}(x, y)$ and $\bar{s_1}(x, y)$ are accurate, they would allow us to reason about the bloat more effectively.

We fit the data (sizes of $2 \times 29 \times 25 = 1450$ object files) against the above two models for all 29 noninlined methods. The results of swap, shown in the middle and right of Figure 19, demonstrate a tight fit. Table 4 lists the model parameters of all noninlined methods along with the associated coefficient of determination, R^2, which quantifies the goodness of the fit. As R^2 is nearly 1 in all cases, we conclude that the measurements consistently support our models. Henceforth we use the models to approximate the size.

Carefully examining the parameters reveals the positive nature of the change induced by generalized hoisting. First, note that the sums of the coefficients of the two trees ($b_0 + d_0$ vs. $b_1 + c_1 + a_1 + d_1$) are similar, as indicated by the "diff" column that shows their difference. These sums are in fact $\bar{s_0}(1, 1)$ and $\bar{s_1}(1, 1)$, reflecting exactly one instantiation of the respective method. The sums should not differ, as they are associated with the same code; our new design has an effect only when more instantiations are created.

Since $s(x, y) \approx b + cx + ay + dxy$, the real goal of the refactoring is to reduce d, the amount of bytes dependent on both C and A. We cannot make these bytes go away. But we can shift them to other parameters; preferably to b (bytes independent of both C and A), but also to c or a (bytes depend on C or A, but not on both). And indeed, comparing d_0 to d_1 in Table 4 reveals that we have successfully done so, as d_1 is significantly smaller than d_0 across all the methods. In Group *i*, the bytes are shifted to b_1, in Group *ii* to a_1, in Groups *iv* and *v* to c_1, and in Group *iii* to both c_1 and a_1.

Let $R(x, y) = \bar{s_0}(x, y)/\bar{s_1}(x, y)$ denote the *bloat ratio*, expressing how much more code is emitted by the original implementation relative to the refactored one. Let us focus on $R(x, 1)$, which reflects the relative price of adding one more comparator. $R(x, 1)$ depends on x, but only up to a point, because systematically increasing x means $R(x, 1)$ converges to $R_c = \lim_{x\to\infty} R(x, 1) = d_0/(c_1 + d_1)$. We thus define R_c to be the *comparator bloat ratio*. We likewise define $R_a = \lim_{y\to\infty} R(1, y) = d_0/(a_1+d_1)$ to be the *allocator bloat ratio*, and we define $R_{ac} = \lim_{x,y\to\infty} R(x, y) = d_0/d_1$ to be the *joint bloat ratio*. The ratios allow us to quantify the effectiveness of the new design in reducing the bloat; they are shown in Figure 20 (same order as in Table 4).

There is no difference between the three ratios of methods in Group *i* (swap etc.), because most bytes have shifted to b_1, and none exclusively depend on C or A. We can see that, asymptotically, the original swap generates 10x more bloat than our refactored version. In Group *ii*, adding a comparator to the original design can be up to 13x more expensive; though adding an allocator is equally expensive (as all the code depends on the allocator even in the refactored design). The comparator and joint ratios are equal in Group *ii*, as $c_1 = 0$. In Groups *iv*–*v*, an added allocator can be up to 25x less expensive with the refactored version. (The allocator/joint ratios are equal because $a_1 = 0$.) Finally, Group *iii* is the only case where the joint ratio is different, since both c_1 and a_1 are nonzero, namely, some bytes exclusively depend on noninlined Rb_cmp code, and some on Rb_alloc code.

6.3 Drawbacks of Generalized Hoisting

Unlike nested classes, which we merely need to move outside, generalized hoisting requires "real" refactoring. Still, the changes we applied to the Rb_tree were trivial, and we believe that they can be applied by average programmers. The technique is certainly suitable for library implementers in terms of their expertise and the cost-effectiveness of their efforts, from which all the library users would benefit.

In our example, there were only three type parameters involved, making the refactoring feasible. More parameters would make things challenging, and we are doubtful whether our approach would scale. We speculate, however, that the principles might still apply, and we believe this subject merits further research. One possible approach might be *externalized hoisting*: Similarly to nested classes, we can move any given member method f to an external scope and replace

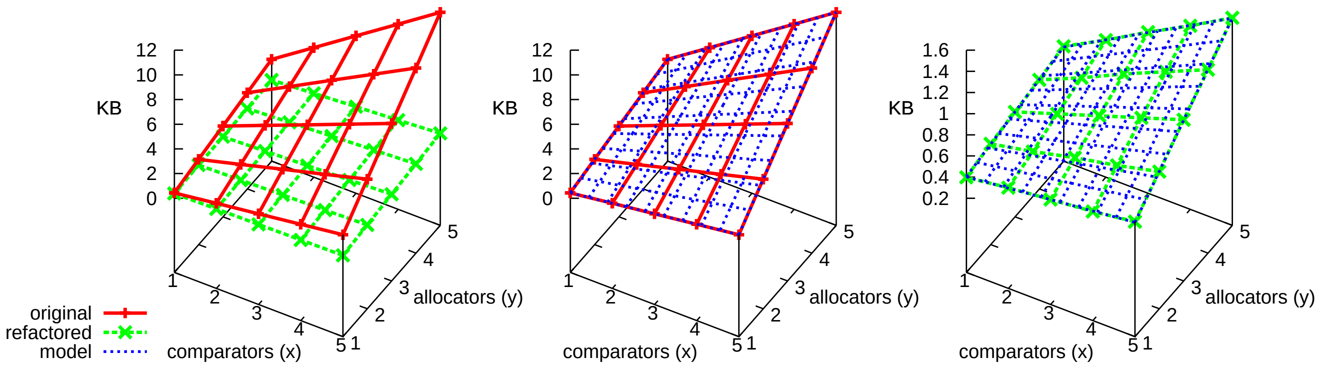

Figure 19. *The size $s(x,y)$ of multiple swap instantiations. Our refactored red-black tree yields nearly an order of magnitude less code relative to the original GCC tree (notice the scale-change in the vertical axis of the rightmost figure). The models of $s(x,y)$ are accurate.*

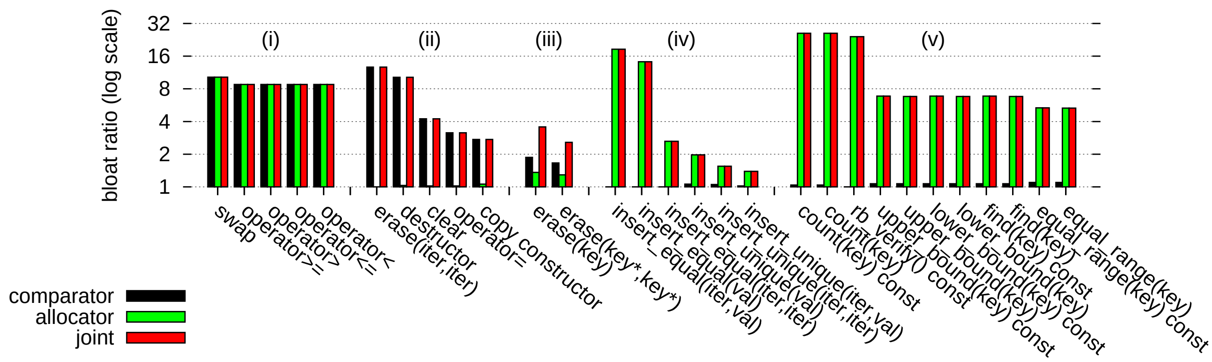

Figure 20. *Comparing the two designs with the comparator (R_c), allocator (R_a), and joint (R_{ca}) bloat ratios.*

it with an inlined version that invokes the now-external function; the type parameter list of the generic now-external f would be minimized to only include its real dependencies. The drawback is losing the reference to "this", and having to supply the relevant data member as arguments.

7. Compiler and Language Support

Some of the bloat reduction achieved through generalized hoisting can be achieved by compiler optimizations. We are aware of one compiler, Microsoft VC++, which employs heuristics to reuse functions that are identical at the assembly level. This, however, does not produce perfect results [4]. But more importantly, such heuristics are inherently limited to methods like those from Group *v* (Table 4), that require no manual modification; all the rest of the Rb_tree methods are different at the assembly level for different type parameters (unless the generalized hoisting technique is applied).

In Section 2, we have presented the conventional solutions to the classic multi-index database problem and noted that they are based on runtime polymorphism. In Section 3 we have utilized SCARY assignments to devise a new solution that is largely based on compile-time polymorphism, and in Section 4 we have shown that this solution is faster. In certain cases, it is possible for the compiler to automatically transform a solution that is based on runtime polymorphism to a solution that is based on compile-time polymorphism. But such transformations would require whole-program optimization, which would make it inapplicable to most real-world C++ programs (which rely on dynamic linking).

Replacing inner classes with aliases and decomposing a class with generalized hoisting can be perceived as "tricks" we must employ since the language does not directly support the notion of minimizing dependencies between the members and parameters of a generic class. Alternatively, we might support a general variant of SCARY in the language and type system by allowing programmers to explicitly specify dependencies for class and method members of a generic type. This would, e.g., be done as briefly illustrated in Figure 21. We intend to investigate this issue in the future.

8. Generalizing to Other Languages

Our findings are applicable to languages that, upon different type parameters, emit different instantiations of the generic class. Such languages can utilize compile-time polymorphism and the aggressive optimizations it makes possible, but at the same time, they are susceptible to bloat.

C# is such a language. Unlike C++, C# emits instantiations at runtime as needed, and if the parameters involved are references (pointer types), the emitted code coalesce to a common instantiation. (This is somewhat similar to the C++

```
template<typename X, typename Y, typename Z> struct C {

  void f1() utilizes X,Z {
    // only allowed to use X or Z, not Y
  }

  void f2() {
     // for backward compatibility, this is
     // equivalent to: void f2() utilizes X,Y,Z
  }

  class Inner_t utilizes Y {
     // only allowed to use Y, not X nor Z
  };
};
```

Figure 21. *With the "utilizes" keyword, programmers would be able to accurately express dependencies; compilers should be able to enforce and exploit this in a straightforward manner.*

void∗ pointer hoisting technique [49].) But if the parameters are "structures" (value types), then a just-in-time (JIT) specialization is emitted, compiled, and optimized, achieving performance almost matching hand-specialized code [36].

C#, however, provides a weaker support to type aliasing. Its "using" directive is similar to C++'s "typedef", but the alias only applies to the file it occurs in. This means that it is currently impossible to hide the fact that the dependencies were minimized and that the nested class was moved outside; users must be made aware and explicitly utilize the now-external type, possibly by changing their code.

We note, however, that support for generic programming is improving. In 2003, Garcia et al. compared languages based on several generics-related desired properties (including type aliasing), and they generated a table that lists which language supports which property [24]. The table entries were 52% "full". This table was revisited in 2007 [25] and in 2009 [47], and became 57% and 84% full, respectively. (We only consider languages that appeared in more than one table version; C#'s "score" was nearly tripled.) It is therefore not unlikely that type aliasing would be added to C# in the future. And this paper provides yet another incentive.

We note in passing that C#'s standard iterators follow the classic design pattern (iterators implement an abstract interface) and hence pay the price of runtime polymorphism; we have shown that the overheads can be significant. However, there is no technical difficulty preventing a C++-like implementation. And, regardless, our findings are general and apply to all generic classes, not just to iterators.

Our ideas also apply to D [5]. If the nested class is static, moving it outside is like doing it in C++, as D supports type aliasing. But unlike C++ and C#, D also supports non-static nested classes, which can access the outer object's members. And so moving such classes outside means breaking this association. While somewhat less convenient, we can resolve this difficulty by manually adding a data member referring to the outer object. This presents the designer with a tradeoff of convenience vs. the benefits detailed in this paper.

Haskell and standard ML are not object oriented languages, but both can represent nested types within generic types [9]. Both languages can be implemented in a way that utilizes multiple instantiations and compile-time polymorphism [34, 52], in which case some of our findings apply (Section 5.1).

Java utilizes generics for compile-time type safety, not compile-time polymorphism. Thus, our results do not apply.

9. Related Work

In statically-typed languages like C++, Java, and C#, the use of runtime polymorphism translates to indirect branches, where addresses of call targets are loaded from memory. In the early 1990s, Fisher argued that indirect function calls "are unavoidable breaks in control and there are few compiler or hardware tricks that could allow instruction-level parallelism to advance past them" [22]. Not only does indirect branching prevent inlining, but it also hinders opportunities for other optimizations such as better register allocation, constant folding, etc. [6]. In addition, pipelined processors are bad at predicting such branches, inflicting pipeline flushes that further degrade the performance [33]. Consequently, the problem is the focus of numerous studies.

"Devirtualization" attempts to transform the indirect calls of a program to direct calls. Static devirtualization, with whole program optimizers, was applied to language like C++ [6, 3] and Modula-3 [14]. But in recent years a lot of effort has been put into dynamic devirtualization in the context of Java and JIT compiling. The function call graph is inferred at runtime [2, 54], and, when appropriate, such information is used for inlining devirtualized calls [13, 1, 32, 27]. (This work is potentially applicable to also C# and the .NET environment.) In parallel, architecture researchers have designed indirect branch predictors in an attempt to elevate the problem [45, 37], and such specialized hardware is currently deployed in state-of-the-art processors, like the Intel Core2 Duo [28]. Despite all this effort, the problem is consistent and prevalent [7, 18, 39, 54, 33].

Compile-time polymorphism attacks the problem in its root cause, by avoiding indirect branches. It is explicitly designed to allow generic code to achieve performance comparable to that of hand-specialized code [48], a goal that is often achieved [51, 35, 36, 26]. The programming technique we propose makes compile-time polymorphism applicable to a wider range of problems. To exemplify, we have shown how to utilize the prevalent classic iterator design pattern [23] in a way that nearly eliminates indirect branching.

Another problem that has spawned much research is executable compaction [3, 11, 30]. Section 5.1 described template hoisting [8], which is the dominant programming technique to reduce code bloat caused by generic programming. We have generalized this technique to reduce the bloat further. Bourdev and Järvi proposed an orthogonal technique involving metaprogramming and manual guidance [4].

10. Conclusions

We advocate a design principle whereby the dependencies between the members and the type parameters of a generic class should be minimized, we propose techniques to realize this principle, and we show that the principle can be leveraged to achieve faster and smaller programs.

Generic programming is utilized by several languages to produce more efficient code. The full compile-time knowledge regarding the types involved allows for compile-time polymorphism, which obviates the need for dynamic binding and enables aggressive optimizations such as inlining. But the applicability of compile-time polymorphism is inherently limited to homogeneous settings, where the types involved are fixed. When programmers need to handle a set of heterogeneous types in a uniform manner, they typically have to introduce an abstraction layer to hide the type differences. They therefore resort to traditional runtime polymorphism through inheritance and virtual calls, hindering the aforementioned optimizations.

We show that the homogeneity limitation is not as constraining as is generally believed. Specifically, we target inner classes that nest in a generic class. We make the case that instantiating the outer class multiple times (with multiple type parameters) does not necessarily mean that the types of the corresponding inner classes differ. We demonstrate that the resulting interchangeability of the latter can be exploited to produce faster code. We do so by utilizing the canonical iterator design pattern (which heavily relies on dynamic binding) in a novel way that entirely eliminates dynamic binding from the critical path. We evaluate the proposed design and demonstrate a x1.2 to x2.1 speedup. While our example concerns C++/STL iterators, our ideas are applicable to any generic class within any programming language that realizes genericity with multiple instantiations (such as C# and D).

We find that, for programmers, obtaining the runtime speedups is nearly effortless and only requires to use the language in a previously unconceived manner ("SCARY assignments") that exploits the interchangeability. But for this to be possible, two conditions must be met. The first is technical. The designers of a generic class should carefully consider the relationship between its type parameters and its nested classes; if an inner class does not depend on all the type parameters, it should be moved outside and be replaced with an alias that minimizes the dependencies. This makes SCARY assignments legal under existing, unmodified compilers. The designers should then declare that no other dependencies exist and thereby allow users to safely exploit the consequent interchangeability. We thus propose to amend standard APIs like STL to reflect the suggested principle; the change will not break old code, but rather, allow for a "new" kind of code.

The second condition is overcoming the typical initial reaction of programmers when presented with SCARY assignments, finding it hard to believe that such assignments conform to the type system and, if so, are a useful technique. In our experience, it is easy to change their minds.

A positive outcome of minimized dependencies is reduced code bloat, as less algorithm instantiations are needed (regardless of whether SCARY assignments are utilized). We suggest a "generalized hoisting" programming paradigm that generalizes this principle in order to further reduce the bloat. By this paradigm, a generic class is decomposed into a hierarchy that minimizes the dependencies between its generic type parameters and *all* of its members (inner classes *and* methods), without introducing indirection that degrades performance. We apply this technique to GCC's STL and obtain up to 25x reduction in object code size. Similarly to our above suggestions, the technique is useful for languages that realize genericity with multiple instantiations.

We have submitted a proposal [38] to the ISO C++ committee to change the standard to reflect our ideas in the upcoming C++ revision, C++0x.

Acknowledgments

We thank the anonymous reviewers for their insightful comments. The first author also thanks Kai-Uwe Bux (Cornell U.), Dilma Da Silva (IBM), Ronald Garcia (Rice U.), Robert Klarer (IBM) Uri Lublin (Redhat), Nate Nystrom (IBM), David Talby (Amazon), Michael Wong (IBM), Amiram Yehudai (TAU), and Greta Yorsh (IBM) for providing much appreciated feedback. Finally, and most importantly, the first author thanks Ziv Balshai (Intel) for asking the original question that started the whole thing: "Why does this code compile under Linux but not under Windows?" Hopefully, the code will soon compile on both.

References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. F. Sweeney, "*Adaptive optimization in the Jalapeńo JVM*". In 15th *ACM Conf. on Object Oriented Prog. Syst. Lang. & App. (OOPSLA)*, pp. 47–65, 2000.

[2] M. Arnold and D. Grove, "*Collecting and exploiting high-accuracy call graph profiles in virtual machines*". In *IEEE Int'l Symp. Code Generation & Optimization (CGO)*, pp. 51–62, 2005.

[3] D. F. Bacon and P. F. Sweeney, "*Fast static analysis of C++ virtual function calls*". In 11th *ACM Conf. on Object Oriented Prog. Syst. Lang. & App. (OOPSLA)*, pp. 324–341, 1996.

[4] L. Bourdev and J. Järvi, "*Efficient run-time dispatching in generic programming with minimal code bloat*". In *Symp. on Library-Centric Software Design (LCSD)*, Oct 2006.

[5] W. Bright, "*D programming language*". http://www.digitalmars.com/d

[6] B. Calder and D. Grunwald, "*Reducing indirect function call overhead in C++ programs*". In 21st *ACM Symp. on Principles of Prog. Lang. (POPL)*, pp. 397–408, 1994.

[7] B. Calder, D. Grunwald, and B. Zorn, "*Quantifying behavioral differences between C and C++ programs*". *J. Prog. Lang.* **2**, pp. 313–351, 1994.

[8] M. D. Carroll and M. A. Ellis, *Designing and Coding Reusable C++*. Addison-Wesley, 1995.

[9] M. M. T. Chakravarty, G. Keller, S. P. Jones, and S. Marlow, "*Associated types with class*". In 32nd *ACM Symp. on Principles of Prog. Lang. (POPL)*, pp. 1–13, Jan 2005.

[10] S. Chapin et al., "*Benchmarks and standards for the evaluation of parallel job schedulers*". In 5th *Workshop on Job Scheduling Strategies for Parallel Processing (JSSPP)*, pp. 67–90, Springer-Verlag, Apr 1999. Lect. Notes Comput. Sci. vol. 1659.

[11] D. Citron, G. Haber, and R. Levin, "*Reducing program image size by extracting frozen code and data*". In *ACM Int'l Conf. on Embedded Software (EMSOFT)*, pp. 297–305, 2004.

[12] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT Press, 2nd ed., 2001.

[13] D. Detlefs and O. Agesen, "*Inlining of virtual methods*". In *European Conf. on Object-Oriented Prog. (ECOOP)*, pp. 258–278, 1999.

[14] A. Diwan, K. S. McKinley, and J. E. B. Moss, "*Using types to analyze and optimize object-oriented programs*". *ACM Trans. on Prog. Lang. & Syst. (TOPLAS)* **23(1)**, pp. 30–72, 2001.

[15] J. J. Dongarra, H. W. Meuer, H. D. Simon, and E. Strohmaier, "*Top500 supercomputer sites*". http://www.top500.org/

[16] C. L. Dumitrescu and I. Foster, "*GangSim: A simulator for grid scheduling studies*". In 5th *IEEE Int'l Symp. on Cluster Comput. & the Grid (CCGrid)*, pp. 1151–1158 Vol. 2, 2005.

[17] A. Duret-Lutz, T. Géraud, and A. Demaille, "*Design patterns for generic programming in C++*". In 6th *USENIX Conf. on Object-Oriented Technologies (COOTS)*, p. 14, 2001.

[18] M. A. Ertl, T. Wien, and D. Gregg, "*Optimizing indirect branch prediction accuracy in virtual machine interpreters*". In *ACM Int'l Conf. Prog. Lang. Design & Impl. (PLDI)*, pp. 278–288, 2003.

[19] A. Ezust and P. Ezust, *An Introduction to Design Patterns in C++ with Qt 4*. Prentice Hall, 2006.

[20] D. G. Feitelson, "*Experimental analysis of the root causes of performance evaluation results: a backfill case study*". *IEEE Trans. Parallel Distrib. Syst. (TPDS)* **16(2)**, pp. 175–182, Feb 2005.

[21] D. G. Feitelson, "*Metric and workload effects on computer systems evaluation*". *IEEE Computer* **36(9)**, pp. 18–25, Sep 2003.

[22] J. A. Fisher and S. M. Freudenberger, "*Predicting conditional branch directions from previous runs of a program*". In 5th *Arch. Support for Prog. Lang. & Operating Syst. (ASPLOS)*, pp. 85–95, 1992.

[23] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[24] R. Garcia, J. Järvi, A. Lumsdaine, J. G. Siek, and J. Willcock, "*A comparative study of language support for generic programming*". In 18th *ACM Conf. on Object Oriented Prog. Syst. Lang. & App. (OOPSLA)*, pp. 115–134, Oct 2003.

[25] R. Garcia, J. Järvi, A. Lumsdaine, J. G. Siek, and J. Willcock, "*An extended comparative study of language support for generic programming*". *J. Func. Prog. (JFP))* **17(2)**, pp. 145–205, Mar 2007.

[26] J. Gerlach and J. Kneis, "*Generic programming for scientific computing in C++, Java, and C#*". In 5th *Advanced Parallel Processing Technologies (APPT)*, pp. 301–310, Sep 2003. Lect. Notes Comput. Sci. vol. 2834.

[27] N. Glew and J. Palsberg, "*Type-safe method inlining*". *J. Sci. Comput. Program.* **52(1-3)**, pp. 281–306, 2004.

[28] S. Gochman et al., "*The Intel Pentium M processor: Microarchitecture and performance*". *Intel Technology Journal* **7(2)**, May 2003.

[29] M. Hansen, "*How to reduce code bloat from STL containers*". *C++ Report* **9(1)**, pp. 34–41, Jan 1997.

[30] H. He, J. Trimble, S. Perianayagam, S. Debray, and G. Andrews, "*Code compaction of an operating system kernel*". In *IEEE Int'l Symp. Code Generation & Optimization (CGO)*, pp. 283–298, 2007.

[31] A. Iosup, D. H. Epema, T. Tannenbaum, M. Farrellee, and M. Livny, "*Inter-operating grids through delegated matchmaking*". In *ACM/IEEE Supercomputing (SC)*, pp. 1–12, 2007.

[32] K. Ishizaki, M. Kawahito, T. Yasue, H. Komatsu, and T. Nakatani, "*A study of devirtualization techniques for a Java just-in-time compiler*". In *ACM Conf. on Object Oriented Prog. Syst. Lang. & App. (OOPSLA)*, pp. 294–310, 2000.

[33] J. A. Joao, O. Mutlu, H. Kim, R. Agarwal, , and Y. N. Patt, "*Improving the performance of object-oriented languages with dynamic predication of indirect jumps*". In *Arch. Support for Prog. Lang. & Operating Syst. (ASPLOS)*, pp. 80–90, 2008.

[34] M. P. Jones, "*Dictionary-free overloading by partial evaluation*". *Lisp and Symbolic Computation* **8(3)**, pp. 229–248, 1995.

[35] C. E. Kees and C. T. Miller, "*C++ implementations of numerical methods for solving differential-algebraic equations: design and optimization considerations*". *ACM Trans. Math. Softw.* **25(4)**, pp. 377–403, 1999.

[36] A. Kennedy and D. Syme, "*Design and implementation of generics for the .NET common language runtime*". In *ACM Int'l Conf. Prog. Lang. Design & Impl. (PLDI)*, pp. 1–12, 2001.

[37] H. Kim, J. A. Joao, O. Mutlu, C. J. Lee, Y. N. Patt, and R. Cohn, "*VPC prediction: reducing the cost of indirect branches via hardware-based dynamic devirtualization*". In 34th *Int'l Symp. on Computer Archit. (ISCA)*, p. 2007, 424–435.

[38] R. Klarer, B. Stroustrup, D. Tsafrir, and M. Wong, "*SCARY iterator assignment and initialization*". ISO C++ standards committee paper N2913=09-0103, Jul 2009. Frankfurt, Germany. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2009/n2913.pdf

[39] J. Lau, M. Arnold, M. Hind, and B. Calder, "*Online performance auditing: using hot optimizations without getting burned*". In *ACM Int'l Conf. Prog. Lang. Design & Impl. (PLDI)*, pp. 239–251, 2006.

[40] A. Legrand, L. Marchal, and H. Casanova, "*Scheduling distributed applications: the SimGrid simulation framework*". In *IEEE Int'l Symp. on Cluster Comput. & the Grid (CCGrid)*, p. 138, 2003.

[41] A. Mu'alem and D. G. Feitelson, "*Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling*". *IEEE Trans. Parallel Distrib. Syst. (TPDS)* **12(6)**, pp. 529–543, Jun 2001.

[42] "*Parallel Workloads Archive*". http://www.cs.huji.ac.il/labs/parallel/workload

[43] "*Grid Workloads Archive*". http://gwa.ewi.tudelft.nl

[44] "*Proc(5): process info pseudo-filesystem - Linux man page*". http://linux.die.net/man/5/proc (Acceded Mar 2009).

[45] A. Roth, A. Moshovos, and G. S. Sohi, "*Improving virtual function call target prediction via dependence-based pre-computation*". In 13th *ACM Int'l Conf. on Supercomput. (ICS)*, pp. 356–364, 1999.

[46] E. Shmueli and D. G. Feitelson, "*On simulation and design of parallel-systems schedulers: are we doing the right thing?*". *IEEE Trans. on Parallel Distrib. Syst. (TPDS)* **20(7)**, pp. 983–996, Jul 2009.

[47] J. G. Siek and A. Lumsdaine, "*A language for generic programming in the large*". *J. Science of Comput. Programming*, 2009. To appear.

[48] A. Stepanov, "*The standard template library: how do you build an algorithm that is both generic and efficient?*". *Byte* **10**, Oct 1995.

[49] B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, 3rd ed., 1997.

[50] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "*Backfilling using system-generated predictions rather than user runtime estimates*". *IEEE Trans. Parallel Distrib. Syst. (TPDS)* **18(6)**, pp. 789–803, Jun 2007.

[51] T. L. Veldhuizen and M. E. Jernigan, "*Will C++ be faster than Fortran?*". In *Int'l Sci. Comput. in Object-Oriented Parallel Environments (ISCOPE)*, 1997.

[52] S. Weeks, "*Whole-program compilation in MLton*". In *Workshop on ML*, p. 1, ACM, 2006.

[53] Wikibooks, "*C++ programming/code/design patterns*". http://en.wikibooks.org/wiki/C++_Programming/Code/Design_Patterns.

[54] X. Zhuang, M. J. Serrano, H. W. Cain, and J-D. Choi, "*Accurate, efficient, and adaptive calling context profiling*". In *ACM Int'l Conf. Prog. Lang. Design & Impl. (PLDI)*, pp. 263–271, 2006.