Sifting out the Mud: Low Level C++ Code Reuse

Bjorn De Sutter   Bruno De Bus   Koen De Bosschere
Electronics And Information Systems Department
Ghent University, Belgium
{brdsutte,bdebus,kdb}@elis.rug.ac.be

ABSTRACT

More and more computers are being incorporated in devices where the available amount of memory is limited. This contrasts with the increasing need for additional functionality and the need for rapid application development. While object-oriented programming languages, providing mechanisms such as inheritance and templates, allow fast development of complex applications, they have a detrimental effect on program size. This paper introduces new techniques to reuse the code of whole procedures at the binary level and a supporting technique for data reuse. These techniques benefit specifically from program properties originating from the use of templates and inheritance. Together with our previous work on code abstraction at lower levels of granularity, they achieve additional code size reductions of up to 38% on already highly optimized and compacted binaries, without sacrificing execution speed. We have incorporated these techniques in Squeeze++, a prototype link-time binary rewriter for the Alpha architecture, and extensively evaluate them on a suite of 8 real-life C++ applications. The total code size reductions achieved post link-time (i.e. without requiring any change to the compiler) range from 27 to 70%, averaging at around 43%.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*C++*; D.3.4 [**Programming Languages**]: Processors—*code generation; compilers; optimization*; E.4. [**Coding and Information Theory**]: Data Compaction and Compression—*program representation*

General Terms

Experimentation, Performance

Keywords

Code compaction, code size reduction

1. INTRODUCTION

More and more computers are being incorporated in devices where the available amount of memory is limited, such as PDAs, set top boxes, wearables, mobile and embedded systems in general. The limitations on memory size result from considerations such as space, weight, power consumption and production cost.

At the same time, ever sophisticated applications have to be executed on these devices, such as encryption and speech recognition, often accompanied by all kinds of eye-candy and fancy GUIs. These applications have to be developed in shorter and shorter design cycles. More complex applications, i.e. providing more functionality, generally mean larger applications. Additional functionality is however not the only reason why applications are becoming bigger. Another important reason is the use of modern software engineering techniques, such as OO-frameworks and component-based development, where generic building blocks are used to tackle the complexity problem. These building blocks are primarily developed with reusability and generality in mind. An application developer often uses only part of a component or a library, but because of the complex relation between these building blocks and the straightforward way linkers build a program (only using symbolic references), the linker often links a lot of redundant code and data into an application. Even useful code that is linked with the application will often involve some superfluous instructions, since it was not programmed with that specific application context in mind.

During the last decade, the creation of smaller programs using compaction and compression techniques was extensively researched. The differences between the two categories is that, while compressed programs need to be decompressed before being executed, compacted programs are directly executable.

At the hardware side, the techniques used range from architectural design to hardware supported dynamic decompression. Examples are the design and wide-spread use of the code size efficient Thumb ISA [36] and dynamic decompression when code is loaded into higher levels of the memory hierarchy [22, 35].

At the software side, a wide range of techniques is developed. Whole-program analysis and code extraction avoid to some extent the linking of redundant code with a program [1, 32, 33]. Application-specific, ultra compact instruction sets are interpreted [18, 19] and frequently repeated instruction sequences within a program are identified and abstracted into procedures [6, 17] or macro-operations [4]. Dynamic decompression is done in software [9] as well. Charles Lefurgy's Phd. thesis [24] provides an excellent overview of these techniques.

As object-oriented programming languages (OOPL) engage the programmer to develop and use reusable code, it is no surprise that the average application written in OOPL contains a considerable amount of redundant code. (For a set of Java programs, Tip and Palsberg [34] on average found more than 40% of all methods to be unreachable.) Besides this, the facilities provided by OOPL to develop reusable code at the source code level, such as template instantiation and class inheritance, often result in duplicate code fragments at the assembly level, thus needlessly increasing program size again. For each different template instantiation, e.g., a separate specialization or low-level implementation is generated. While at the source code level the instantiations have different types, their low-level implementations are often very similar. Pointers to different types of object, e.g., have a different type at the source code level, but are all simple addresses at the lower level.

Because of the high overhead in code size, the use of modern software engineering techniques and OOPL is often not even considered viable for embedded or portable applications. One can ask whether the limitations on complexity and offered functionality of these applications result from hardware limitations (memory size, power consumption, etc.) or from the overhead introduced when trying to manage the complexity using modern software engineering techniques. This question and the problems with code size have led to the development of Embedded C++ [11], a subset of the C++ language from which features such as templates that lead to code bloat have been removed.

The techniques introduced and discussed in this paper address the mentioned code bloat directly. Without sacrificing language features (and not requiring any changes to compilers), programs are produced that are both significantly smaller and at the same time faster. As these programs are still directly executable, all forms of run-time optimization and dynamic (de)compression techniques can still be applied on them. This is all achieved by aggressive whole-program optimization and extensive code reuse after a program has been linked. The achieved code size reductions range from 27 to 70%, averaging around 43%. From a commercial point of view, this work leads to somewhat less than a halved code size requirement, which under otherwise identical conditions (e.g. a fixed maximum die area available for ROM) means a year and a half earlier on the market.

In the whole picture we sketched so far, the linker, albeit a very important component in any software development environment, is largely understudied. Several techniques are being used to avoid linking multiple identical specializations into a program. These techniques include incremental linking and the use of repositories [25], which rely on compiler-linker cooperation. They are based on the forwarding of extra information by the compiler to the linker or on feedback from the linker to the compiler. However, these techniques do not at all address the possible reuse of nearly identical code fragments or code fragments that are identical only at the assembly level, but not at the source level.

In the past we have proposed applying code and data compaction in a link-time binary rewriter named Squeeze. We have demonstrated how aggressive whole-program optimization [10] and combined redundant data and code elimination [7] at link-time (or more precisely post link-time) can effectively reduce the size of programs. It was no surprise that these techniques and especially the elimination of unreachable code by detecting statically allocated data (including procedure pointers) that is dead, performed much better on C++ programs than on C programs. This follows from the discussion above. In [10] we also discussed code abstraction at low levels of granularity: code regions and basic blocks. These code abstraction techniques were evaluated on a set of C programs, and resulted in a significant, but rather limited additional code compaction.

Today we present Squeeze++, our latest prototype link-time program compactor. Its name was chosen, not only because it is a better Squeeze, but because it incorporates a number of new code abstraction and parameterization techniques specifically aimed at reusing whole identical or nearly identical procedures, as typically found in programs written in object-oriented languages such as C++. These new code reuse techniques are introduced in this paper, and our whole

stk.h:

```
template<class T> class Stk {
private:
  static int total_space;
  T* stk;
  int size;
  int elements;
  T* top;
public:
  Stk(void);
  void Push(T elem);
  T Pop(void);
  ~Stk(void);
};
```

stk.cpp:

```
template <class T> T Stk<T>::Pop(void) {
  if (elements==size){
    size-=10;
    total_space-=10;
    stk=(T*) realloc(stk,size*sizeof(T));
    top = &(stk[elements]);
  }
  elements--;
  return *--top;
}
```

main.cpp:

```
#include <stdio.h>
#include "stk.h"

main(){
  Stk<long int> x;
  Stk<int> y;
  x.Push(1000L);
  y.Push(1000);
  printf("%ld %d\n",x.Pop(),y.Pop());
}
```

Figure 1: Example C++ code

range of previous work on code abstraction is reevaluated for a number of C++ programs, showing that code abstraction and parameterization is a very powerful technique for additional code size reduction for C++ programs (up to 38%) on top of aggressive post link-time program optimization. This results in a total link-time code size reduction of up to 70%, which is considerably more than what we previously achieved for C or Fortran programs [10], thus addressing the problems of code bloat and program size where it is most needed: where (redundant) code is most reused.

The remainder of this paper is organized as follows. In section 2 a motivating example is studied. The new code reuse techniques are discussed in section 3. Our previous work on code abstraction techniques is discussed in section 4. Some new insights and enhancements are discussed as well. An extensive evaluation of the discussed techniques is made in section 5. Related work is discussed in section 6, after which our conclusions are drawn in the last section.

2. MOTIVATING EXAMPLE

We have depicted some C++ code fragments in Figure 1. This code is not from a real-world application. It serves our purpose however, as it contains some typical code aspects found in real-world applications.

A template class Stk<T> implements a stack data structure, for different types of objects of type T. The elements on the stack are stored in a private array stk with size elements. The top data member points to the top element of the stack. The number of elements on the stack is stored in elements. The array stk is dynamically allocated. It grows and shrinks in chunks of 10 elements, as objects are pushed or popped. The private static data member total_space is a class member that records the total memory size for one type of stack. Its only purpose is to serve our discussion (although it might be useful for debugging purposes). In the main program, two stacks are created, one for objects of type int and one for objects of type long int.

In Figure 2 we have depicted the control flow graphs (CFGs) and assembly code for the two generated instances of the Pop() method. This code was generated using the Compaq C++ V6.3-002 for Compaq Tru64 UNIX V5.1 compiler for the Alpha platform. The CFGs are generated from the internal representation of the program after SQUEEZE++ has compacted the program. Inlining was not applied since we want to depict these methods isolated from their calling contexts for the purpose of our example. Neither have we applied code abstraction, as we want to show where opportunities for code abstraction and parameterization are found. We have depicted assembly code with more generally understandable mnemonics instead of the Alpha mnemonics, although there is no need to understand this code to understand the discussion of this example.

These CFGs and assembly code obviously are very similar to one another. There are some differences however, indicated by the use of a bold and italic typeface. These differences have two origins:

1. A first difference has to do with the static total_space data member. Each of the specializations of the Stk class has a separate instance of this class member. In the final program, these two instances are stored in the statically allocated data. They are allocated at different locations however, so the locations from which their values are loaded in the Pop method differ. These different locations result in different indices (or relative addresses) in the instructions that access them (index 0x2280 in int Stk<int>::Pop() and index 0x2270 in long Stk<long>::Pop()).

   More generally, all statically allocated objects for some class will have different addresses for each specialization. This is not just the case for class members, but also for, e.g., stored initialization values of local variables in methods (even if these values are the same for all specializations).
2. The second difference results from the different sizes of the objects stored on the stack. On our 64-bit target architecture, int variables occupy 4 bytes, while long int variables occupy 8 bytes. Different instructions are used to load and store these values, corresponding to their respective word widths (e.g. load int vs. load long). The pointer arithmetic also differs: indexing is done using different instructions (add int index vs. add long index) or different relative addresses (0x0004 vs. 0x0008).

With this motivating example, we have shown how very similar the assembly code of different specializations of a template can be, and what some of the most important reasons for differences are. A third origin of differences is often found at calls to methods that depend on the type of object the template is specialized for: a sorting method of some container class, e.g., often calls the compare method of the class it was specialized for. Specializations for different classes will usually have exactly the same sorting routine at the assembly level, except for the call to the compare method. If the method called is a virtual method, the address of the vtable from which the address of the actual called method is loaded will be different. This is very similar to the first origin of differences in the motivating example.

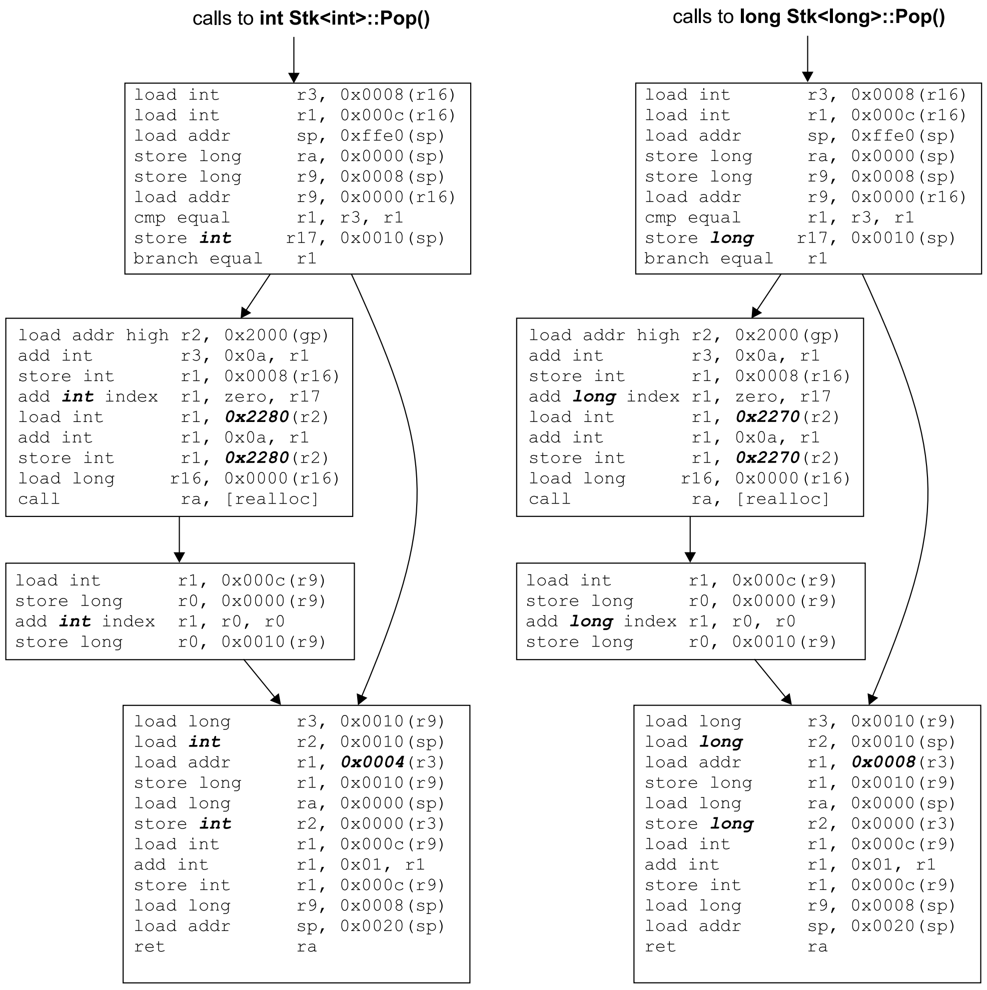
3. REUSING WHOLE PROCEDURES

Programming constructs such as templates and inheritance often result in multiple low-level implementations of source code fragments that show great similarity. Significant compaction of programs can be obtained by abstracting multiple-occurring instruction sequences: a multiple-occurring instruction sequence is put in a separate procedure and all occurrences of the sequence are replaced by calls to that single procedure. This is the inverse of inlining. Looking for multiple-occurring instruction sequences can be done at various levels of granularity, that are discussed in this and the next section. We will look at whole procedures, code regions (i.e. groups of basic blocks with a unique entry and exit point), basic blocks and subblock instruction sequences. In this section we focus on the higher level: that of whole procedures. We also discuss the reuse of statically allocated data, as it creates more opportunities for the code reuse techniques.

3.1 Whole-Procedure Reuse

The targets of whole-procedure reuse are multiple identical procedures that might be found in the program because template specializations are different on the source code level, but identical on the assembly code level.

A question to answer is: what do we consider to be identical procedures? Do we, e.g., consider two procedures identi-



```
load int       r3, 0x0008(r16)
load int       r1, 0x000c(r16)
load addr      sp, 0xffe0(sp)
store long     ra, 0x0000(sp)
store long     r9, 0x0008(sp)
load addr      r9, 0x0000(r16)
cmp equal      r1, r3, r1
store int     r17, 0x0010(sp)
branch equal   r1
```

```
load addr high r2, 0x2000(gp)
add int        r3, 0x0a, r1
store int      r1, 0x0008(r16)
add int index  r1, zero, r17
load int       r1, 0x2280(r2)
add int        r1, 0x0a, r1
store int      r1, 0x2280(r2)
load long     r16, 0x0000(r16)
call           ra, [realloc]
```

```
load int       r1, 0x000c(r9)
store long     r0, 0x0000(r9)
add int index  r1, r0, r0
store long     r0, 0x0010(r9)
```

```
load long      r3, 0x0010(r9)
load int       r2, 0x0010(sp)
load addr      r1, 0x0004(r3)
store long     r1, 0x0010(r9)
load long      ra, 0x0000(sp)
store int      r2, 0x0000(r3)
load int       r1, 0x000c(r9)
add int        r1, 0x01, r1
store int      r1, 0x000c(r9)
load long      r9, 0x0008(sp)
load addr      sp, 0x0020(sp)
ret            ra
```

```
load int       r3, 0x0008(r16)
load int       r1, 0x000c(r16)
load addr      sp, 0xffe0(sp)
store long     ra, 0x0000(sp)
store long     r9, 0x0008(sp)
load addr      r9, 0x0000(r16)
cmp equal      r1, r3, r1
store long    r17, 0x0010(sp)
branch equal   r1
```

```
load addr high r2, 0x2000(gp)
add int        r3, 0x0a, r1
store int      r1, 0x0008(r16)
add long index r1, zero, r17
load int       r1, 0x2270(r2)
add int        r1, 0x0a, r1
store int      r1, 0x2270(r2)
load long     r16, 0x0000(r16)
call           ra, [realloc]
```

```
load int       r1, 0x000c(r9)
store long     r0, 0x0000(r9)
add long index r1, r0, r0
store long     r0, 0x0010(r9)
```

```
load long      r3, 0x0010(r9)
load long      r2, 0x0010(sp)
load addr      r1, 0x0008(r3)
store long     r1, 0x0010(r9)
load long      ra, 0x0000(sp)
store long     r2, 0x0000(r3)
load int       r1, 0x000c(r9)
add int        r1, 0x01, r1
store int      r1, 0x000c(r9)
load long      r9, 0x0008(sp)
load addr      sp, 0x0020(sp)
ret            ra
```

Figure 2: The CFGs generated for int Stk<int>::Pop() and long Stk<long>::Pop().

cal if they are exactly the same, apart from the registers they use for storing local variables? Or apart from the exact instruction schedules? The answer is that we require the procedures to be exactly the same, having the same structure of basic blocks and having exactly the same instructions, in identical schedules, and with identical register operands. This requirement speeds-up the search for identical procedures, as we never need to check whether two variants are functionally equivalent.

One might think that a significant number of functionally equivalent procedures will not be discovered, but this is not the case, as a compiler implementing identical functionality will use identical registers and produce the same basic block layout. There is no reason to assume nondeterminism in this respect. Even for compilers implementing interprocedural register allocation, this is often the case, as template classes are usually defined in separate source code modules and the interprocedural register allocation typically is intramodular.

Looking for fully identical procedures is straightforward: we apply a pairwise comparison of the CFGs of procedures by traversing the procedures in depth-first order, during which their instructions are compared. If two or more identical procedures have been found, one master procedure is selected and all call-sites of the others are converted to call the master procedure. Function pointers stored in the statically allocated data of the program (such as method addresses in vtables) are also converted to all point to the master procedure. Note that the search for identical procedures is an iterative process. As a procedure's callees change, they might become identical.

To limit the number of comparisons necessary, a fingerprinting scheme is used. A procedure's fingerprint consists of its number of basic blocks and instructions, and a string describing the structure of its CFG. This string is constructed using a depth-first traversal, during which characters that denote basic block types are concatenated. The type of a block is the type of its last instruction: there are 'C'all blocks, 'R'eturn block, 'B'ranch blocks, etc. Using the fingerprints, the procedures are partitioned, after which the more comprehensive pairwise comparison is performed per partition of similar procedures. Note that the various fingerprinting schemes discussed in this paper are not fundamental for the code reuse techniques. They are very useful however to limit their computational cost.

3.2 Procedure Parameterization

As shown in the motivating example, it is possible that procedures are nearly identical. One possible solution to reuse the common code fragments is to apply abstraction techniques on smaller code fragments, such as the ones we will discuss in the next section. This can be very expensive however. Suppose we need to separately abstract every basic block but one. This would mean that for each basic block but one a procedure call has to be inserted.

A different approach is to merge nearly identical procedures and to add a parameter to the merged procedure. This parameter is used to select and execute the code fragments corresponding to one of the original procedures where the code differs. The merged procedure gets multiple entry points, one for each of the original procedures, at which the parameter is set. This approach requires two questions to be answered: what do we consider as nearly identical procedures and how do we add a parameter to the merged procedure?

As for whole-procedure reuse, we limit the search for nearly identical procedures to procedures with identical structures of basic blocks. A pre-pass phase again partitions the procedures using a fingerprint based on the number of blocks and the structure and the types of the blocks. The number of instructions is now not used to partition procedures. Procedures with identical structures are then pairwise compared and are given an equivalence score. This score equals the (predicted) code size reduction when two procedures would be merged. Each pair of corresponding basic blocks in the two procedures contributes to the score as follows:

- The number of identical instructions starting from the entry points of the blocks is added to the score. If both blocks are completely identical, this is the final contribution of these blocks to the equivalence score. It indicates that a whole block can be eliminated.
- If the blocks are not identical, the number of identical instructions going backwards from the exit points of the blocks is added to the score. 3 points are subtracted however, as we conservatively assume that 3 instructions are needed to implement the conditional branch that needs to be inserted before the different parts of the blocks.

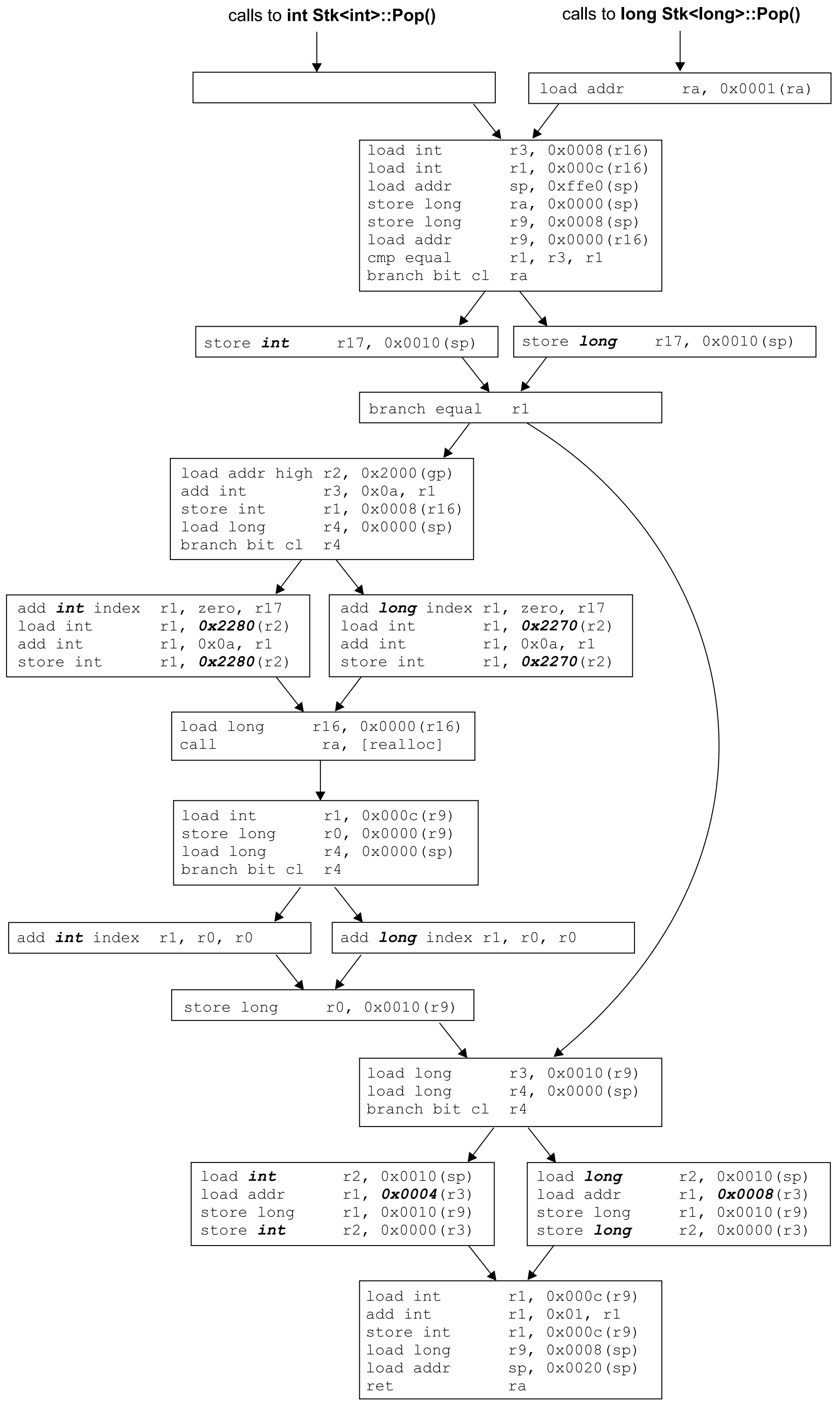
The rationale behind this counting is that identical prefixes and postfixes of instruction sequences in basic blocks will be hoisted or sunk in the merged procedure, thus eliminating one of them. As this counting is only a prediction, and as this is sometimes not very accurate, due to, e.g., possible register renaming or instruction reordering being applied later on during the code compaction, we only count identical instructions in identical orders, with identical operands (register and immediate operands). For nearly identical specializations of the same template class with differences that we discussed in section 2 this suffices. Experiments have shown that the threshold of the equivalence score to apply parameterization is best chosen at 10, i.e. the score has to be at least 10 or more.

Once we have found nearly identical procedures, we must find a location to store the parameter that will be used in the merged procedure to select the correct code to be executed at program points were the original procedures were different. This is not an easy task, because of two observations:

1. We almost always must assume the merged procedure to be re-entrant through possible recursive calls. The reason is that we very often don't know exactly what methods can be called at call-sites with indirect procedure calls. This is in general the case where procedure pointers are used, such as for all virtual method calls. For these indirect calls, we assume that all code addresses stored in the program can be possible targets, resulting in an overly conservative call graph of the program. The consequence of this observation of re-entrancy is that the parameter must be stored on the stack.
2. Transforming the stack behavior of procedures or a program in a link-time or post link-time program compactor is very difficult, if not impossible, due to the general lack of any high-level semantic information about the program being compacted, and particularly about the stack behavior. To rely on compilers to provide this information to the program compactor, would endanger the general applicability of our techniques, so we have chosen to not follow that path. The consequence is that we will not allocate additional space on the stack for storing the parameter.

If we cannot change the stack frame of a procedure, how are we going to store a parameter in it? The solution involves the return address corresponding to procedure calls. If a procedure is not a leaf procedure, this address is definitely stored on the stack, as the calls in the procedure overwrite the return address. If the procedure is a leaf procedure, the return address is often not stored on the stack, but in that case we are sure that the procedure is not re-entrant and we know that at least some register will hold the return address throughout the whole procedure. It is therefore safe to assume that the location where the return address is stored, either in a register or on the stack, is a location that holds its value throughout a whole procedure's execution, whether this execution is interrupted by recursive calls or not. On our target architecture, as on most 32-bit RISC architectures having a fixed instruction width and 4-byte aligned instructions, the 2 least significant bits of the return address are always zero. We can therefore use these 2 bits to encode at most 4 different parameters corresponding to at most 4 procedures being parametrized and merged. The maximum number of procedures that can be merged to a single parametrized procedure is limited to 4, but as we will see in the evaluation section, this is enough to get a significant compaction with procedure parameterization. To select groups of 4 procedures from a larger group of nearly identical procedures, we use a greedy algorithm based on the equivalence scores.

Using the thus stored parameter for conditional branches in the merged procedure body is trivial, as the location of a return address stored on the stack is determined by the calling conventions. The 3 instructions needed for these conditional branches are (1) to load the return address from the stack; (2) to extract the least-significant bits (if necessary) from the return address; (3) the conditional branch on these bits. Note that on architectures where the return instruction doesn't care about the two least-significant bits of the return addresses, there will be no need to remove the parameter from the return address before returning from a call to the merged procedure.


calls to **int Stk<int>::Pop()**
calls to **long Stk<long>::Pop()**

```
load addr      ra, 0x0001(ra)
```

```
load int        r3, 0x0008(r16)
load int        r1, 0x000c(r16)
load addr       sp, 0xffe0(sp)
store long      ra, 0x0000(sp)
store long      r9, 0x0008(sp)
load addr       r9, 0x0000(r16)
cmp equal       r1, r3, r1
branch bit cl   ra
```

```
store int      r17, 0x0010(sp)
```

```
store long     r17, 0x0010(sp)
```

```
branch equal   r1
```

```
load addr high r2, 0x2000(gp)
add int        r3, 0x0a, r1
store int      r1, 0x0008(r16)
load long      r4, 0x0000(sp)
branch bit cl  r4
```

```
add int index  r1, zero, r17
load int       r1, 0x2280(r2)
add int        r1, 0x0a, r1
store int      r1, 0x2280(r2)
```

```
add long index r1, zero, r17
load int       r1, 0x2270(r2)
add int        r1, 0x0a, r1
store int      r1, 0x2270(r2)
```

```
load long     r16, 0x0000(r16)
call           ra, [realloc]
```

```
load int       r1, 0x000c(r9)
store long     r0, 0x0000(r9)
load long      r4, 0x0000(sp)
branch bit cl  r4
```

```
add int index  r1, r0, r0
```

```
add long index r1, r0, r0
```

```
store long     r0, 0x0010(r9)
```

```
load long       r3, 0x0010(r9)
load long       r4, 0x0000(sp)
branch bit cl   r4
```

```
load int        r2, 0x0010(sp)
load addr       r1, 0x0004(r3)
store long      r1, 0x0010(r9)
store int       r2, 0x0000(r3)
```

```
load long       r2, 0x0010(sp)
load addr       r1, 0x0008(r3)
store long      r1, 0x0010(r9)
store long      r2, 0x0000(r3)
```

```
load int        r1, 0x000c(r9)
add int         r1, 0x01, r1
store int       r1, 0x000c(r9)
load long       r9, 0x0008(sp)
load addr       sp, 0x0020(sp)
ret             ra
```

Figure 3: The CFGs for int Stk<int>::Pop() and long Stk<long>::Pop() after parameterization.

For the motivating example in section 2, the final CFGs of the original procedures after parameterization are depicted in Figure 3. As one can see, the total number of instructions in these 2 procedures has gone down from 68 to 56. This 56 includes the number of unconditional branches that need to be inserted because basic blocks can only have one predecessor on a fall-through path. These unconditional branch instructions are not part of our internal program representation, and are hence not depicted.

3.3 Reusing Statically Allocated Data

It often occurs that constant values stored in the read-only data sections of a program occur multiple times. On compiling one source code module, the compiler provides space in the statically allocated data of the generated object file for the addresses of all externally declared objects that are accessed from within the module. While linking and relocating modules, the linker fills in the final addresses. If global objects are accessed from within multiple modules, their address will occur multiple times in the statically allocated data of the final program.

Another case are constant numerical values for which loading them from the data sections is the most efficient way to get them in a register. This is typically the case for floating-point constants such as π or e. These constant values are often stored in the data sections of multiple object files and therefore occur multiple times in the final program.

As data is linked with the program to be loaded somewhere, the result of multiple occurences of identical data is that otherwise identical code fragments can differ in the location from which they load the (identical) data. As our link-time code compactor incorporates a constant propagator that to some extent detects what constant data is loaded, we can convert instructions that load the same constant value from different locations into instructions that load them from the same location. This has proven to be very beneficial for code reuse, and especially for the reuse of whole identical or nearly identical procedures, as it limits the first kind of differences between procedures discussed in section 2. Note however that we apply this conversion independently from the code reuse techniques, as they also improve cache behavior.

4. FINE-GRAINED CODE REUSE

In this section, our previous work on code reuse of more fine-grained identical or functionally equivalent code fragments is rediscussed. On several occasions, new insights and enhancements are introduced.

The use of inheritance and virtual method calls results in procedures that are often not identical or even similar, but still contain similar code fragments, simply because they provide similar functionality. It therefore often happens that parts of such methods are identical or at least functionally equivalent (i.e. the apply the same computations, but on different register operands). Sometimes template instantiations are not similar enough according to our equivalence score metric, but still contain similar or identical code fragments. In both cases, identical or functionally equivalent code fragments can be abstracted into procedures.

The execution of thus abstracted procedures ends at a return instruction, after which the execution of the program always continues at the instruction following the call-site of the abstracted procedure. Therefore, abstracted code fragments must have a unique entry point and a unique exit point. This by definition is the case for basic blocks and subblock instruction sequences. Code fragments consisting of multiple basic blocks can also have a unique entry and exit point. Those fragments are called code regions, and we'll start our discussion of fine-grained code reuse techniques with code regions.

4.1 Code Region Abstraction

The exit and entry points of a code region correspond to a pair of dominator and postdominator blocks. Such pairs therefore uniquely identify a code region. Again a fingerprinting system is used to prepartition all the regions in a program. As the number of code regions is much higher than the number of procedures and as techniques such as register renaming to make functionally equivalent regions using different register operands identical are computation expensive, we limit our search space to fully identical regions: i.e. with identical structure, instructions, schedule and register operands.

Allocating space for the return address poses a problem when procedures are being called from within the abstracted procedure, as we discussed in section 3.2: calls to abstracted code need to store a return address somewhere and this return address can be seen as some kind of parameter. A difference with the discussion in section 3.2 is that, because compiler generated stack allocation mostly takes place in the entry and exit blocks of a procedure, and not in the code in between, we can expect that fewer stack allocations in the original code regions will interfere with the additional stack space we want to allocate. However, while we have found a very conservative way to allocate an additional location to store the return address for such abstracted code regions, it is so conservative that we can almost never apply it and therefore almost never are able to abstract code regions in which procedure calls occur.

One exception is when the code regions to be abstracted end with a return instruction. As the call to the abstracted code in this case is a tail call, there is no need for a procedure call to the abstracted procedure. Applying tail-call optimization, we can just jump into the abstracted procedure and the return at the end of it will return directly to the callers of the procedures from which the region was abstracted.

4.2 Basic Block Abstraction

Like we said before, basic blocks are by definition code fragments with a unique entry and a unique exit block. As opposed to whole procedures with identical functionality, basic blocks with identical functionality show much more variation in the used register operands. The reason is that the compiler allocates registers for a whole procedure. Therefore, the register operands used in a basic block largely depend on the registers used in the surrounding code. For this reason, and because local register renaming (within a single basic block) is not nearly as complex and time consuming as global register renaming (over multiple basic blocks), we'll try to make functionally equivalent basic blocks identical by renaming registers.

Like we did for the other reuse techniques, basic blocks are first partitioned to speed up the detection of equivalent blocks. The fingerprints used to do so include the opcodes of the instructions, but not the register operands. Within each partition, the search for functionally identical blocks is done as follows:

- As long as there are multiple blocks in a partition, a master block is selected.
- We try to rename all the other (slave) blocks to the master block. If a slave block already is identical to the master block, renaming is of course unnecessary.
- The master block and all slave blocks (made) identical to the master block are abstracted into a procedure, and procedure calls to this abstracted procedure replace the original occurrences of the blocks.
- If no slave blocks are found that are or can be made identical to the master block, the master block is removed from the partition.

This is all similar to our previous work in [10]. We have however enhanced the renaming algorithm. Our new renaming algorithm works in three phases:

1. *Comparing the dependency graphs*
   First, the dependency graphs[1] of the master and slave blocks are compared. Conceptually this is done by converting the blocks to a symbolic static single assignment representation and by just comparing whether the symbolic register operands of each instruction are the same.

   One detail of this conversion and comparison is important to discuss: all externally defined registers are converted to a single symbolic register. Consider the example code in Figure 4. The fourth instruction in the master block has two different externally defined source register operands: r7 and r8. In the slave block, register r7 is used twice. In our symbolic representation for comparing the dependency graphs of the two blocks, the same symbolic register x is used for r7 and r8.

2. *Adding copy operations*
   If the slave and master blocks implement the same dependency graph as discussed in the first phase, we try to insert copy operations. These are added for three reasons:

   (a) When externally defined registers differ, copy operations are inserted before the slave block (copy operation (2) in the example).

   (b) When different registers are defined in the blocks that are live[2] at the end of the slave block, copy operations are inserted after the slave block (copy operation (3) in the example).

   (c) When a register is defined in the master block that is not defined in the slave block, but which is live over the whole slave block, copy operations are inserted before and after the slave block to temporarily store the register content in another register. Assume register r0 to be live over the whole slave block in the example. Then two copy operations need to be inserted: (1) and (4).

3. *Actual renaming and abstraction*
   The insertion of copy operations is considered successful if there are no copy operations necessary before or after the slave block that need to copy different registers to the same destination register. If the number of inserted copy operations is small enough for the blocks to be abstracted, the slave block is simply replaced by the master block, thus effectively renaming the slave block. In the example, renaming is possible, but not considered worthwhile, as the number of added copy operations is higher than the code size reduction obtained by abstracting the basic blocks. Note that we also would have to add two call instructions and one return instruction to abstract the blocks.

[1]A dependency graph is a directed graph where the operands of instructions are vertices. Directed edges connect definitions (i.e. write operations) of destination operands with consuming source operands (i.e. operands of operations reading the value produced at the definition).

[2]A register is considered live if its content can be used by some instruction later on during the program execution. If a register is live at some program point we cannot overwrite its value, unless its original value is stored somewhere else (spilled on the stack or in some other register).

The difference between this renaming algorithm and the one we previously published [10] is that the old algorithm combined phases 1 and 2: the copy operations were collected during the dependency graph comparison. To be able to do this in one phase, all externally defined registers used in the blocks must get unique symbolic names (x1, x2, etc. instead of a single x). If this is done, the fourth instruction in the master and slave blocks of the example in Figure 4 have different operands, and the dependency graph is not considered identical. These blocks were therefore not considered renameable with the old algorithm. One could say that the old algorithm could only rename functionally equivalent blocks, whereas the new algorithm is also able to rename blocks to functionally superior blocks.

Cooper and McIntosh [6] propose renaming basic blocks by globally renaming registers instead of inserting register copy operations. This has the disadvantage that renaming to make one pair of blocks identical can affect (even undo) renaming for another pair of blocks. We feel that inserting copy operations is favorable, especially since a separate copy elimination phase after the code abstraction will be able to eliminate most, if not all, of the copy instructions in those cases where global renaming could have been applied to make the blocks identical.

Another advantage of using copy instructions is that these can easily be used to "rename" immediate operands to registers. Just like a copy instruction is inserted, an inserted instruction can store a value in a register that is than used in the abstracted block instead of the immediate operands in the original blocks. This of course is useful only when the corresponding immediate operands or registers in the original blocks differ.

The renaming algorithm is the only transformation we apply to make functionally equivalent (or superior) blocks identical. A question now arises concerning our search space: do the schedules (or order) of the instructions in the master and slave blocks matter?

What if two blocks are identical, except for the order of the instructions? We have done some experiments to answer this question by inserting a pre-pass scheduler, that generates a deterministic schedule given a dependency graph. This means that two blocks with identical dependency graphs but

```
   master            slave         |   master          slave        |   master            slave
                                   |                                |                     r9 := r0           (1)
                                   |                                |                     r8 := r7           (2)
                                   |                                |
r4 := r2 + r1    r1 := r2 + r1     | i1 := x + x    i1 := x + x     | r4 := r2 + r1     r4 := r2 + r1
r4 := r4 - r3    r4 := r1 - r3     | i2 := i1 - x   i2 := i1 - x    | r4 := r4 - r3     r4 := r4 - r3
r0 := r4 / r6    r9 := r4 / r6     | i3 := i2 / x   i3 := i2 / x    | r0 := r4 / r6     r0 := r4 / r6
r4 := r7 * r8    r5 := r7 * r7     | i4 := x * x    i4 := x * x     | r4 := r7 * r8     r4 := r7 * r8
                                   |                                |
                                   |                                |                     r5 := r4           (3)
                                   |                                |                     r0 := r9           (4)

       Original blocks               Using symbolic registers              Renamed blocks
```

Figure 4: Example basic blocks for basic block abstraction.

with different instruction orders are transformed in blocks with the same order. This allows us to disregard scheduling differences when comparing dependency graphs or inserting copy instructions. We have learned that this pre-pass does not yield a significant increase in the number of blocks being abstracted (less than 0.1%). We conjecture that the local schedules generated by the compiler are pretty deterministic.

4.3 Subblock Instruction Sequence Abstraction

Our previous research on abstracting partially matched basic blocks involved two important special cases: saves and restores for callee-saved registers. Most of the time the saves occur in the entry block of a procedure and the restores occur in blocks ending with a return instruction. As this limits the number of code fragments that have to be compared, we can afford the time to try to make these sequences identical by locally rescheduling code (i.e. within the entry or return blocks of a procedure). This rescheduling might be necessary when the compiler has mixed the save and restore instruction sequences with other instructions. While abstracting register save sequences involves some overhead because of the extra call and return, this is not the case for register restores, as tail-call optimization can often eliminate the overhead. How this is done is discussed in more detail in [10].

Whereas we previously found that a more general abstraction of partially matched blocks is computationally quite expensive without offering significant code size reductions for C programs, this is not the case for C++ programs. The reason is again that C++ programs offer more possibilities for code reuse. However we still need to keep the computational cost reasonable, so we restrict the general abstraction of partially matched basic blocks to identical instruction sequences. No register renaming or rescheduling is applied to create identical sequences. Even then an enormous number of instruction sequences have to be compared. A quite efficient way we found to do this is as follows.

A separate pass over the whole code is applied for all possible instruction sequence lengths we want to abstract, starting with the longest sequences and ending with the shortest one (3 instructions). During each pass for instruction sequences with length n

1. all occurring sequences of length n are collected and prepartitioned according to a fingerprint (that now includes the register operands used);

2. for each partition, identical sequences are collected. These multiple identical sequences are placed in separate basic blocks, by splitting up the blocks in which they first occurred.

After this splitting of basic blocks for all sequence lengths, the basic block abstraction technique is applied again to actually abstract the identical sequences in the split basic blocks. By not collecting instruction sequences in phase 1 that were separated in phase 2 of a previous pass, larger blocks are greedily abstracted first. Although this is not optimal, (more beneficial splitting of basic blocks might be possible), we found this to be a good compromise between computational efficiency and code size reductions. By having a separate pass for all possible instruction sequence lengths, the memory-footprint can be kept acceptable without sacrificing execution speed. Note that for high values of n, where large fingerprints need to be used, the number of instruction sequences in a program decreases, as basic blocks on average consist of 4–5 instructions only.

4.4 Summary

Reusing multiple instances of identical code fragments can be done at various levels of granularity: procedures, code regions with unique entry and exit points, basic blocks and partially matched basic blocks. Looking at the ratio between computational cost and code size reduction certain trade-offs have to be made. Register renaming to create identical code fragments is applied at the basic block level only. Rescheduling to create identical code sequences is applied for callee-saved register stores and restores only.

It is obvious that the different abstraction levels influence each other: without procedural abstraction or parameterization, all the identical blocks or regions in procedures will still be abstracted if basic block and region abstraction are applied. The introduced overhead will then be larger however, since a procedure is then largely replaced by a number of calls to abstracted procedures.

5. EXPERIMENTAL EVALUATION

The algorithms described in this paper are implemented in SQUEEZE++, a binary rewriting tool aiming at code compaction for the Alpha architecture. The Alpha architecture was chosen because of its clean nature that eases the implementation of algorithms, in particular of the SQUEEZE++ backend. We firmly believe that the algorithms implemented in SQUEEZE++ are generally applicable and not architecture specific, and therefore consider SQUEEZE++ and the results obtained with it a valid proof-of-concept.

5.1 The Benchmarks

The performance of SQUEEZE++ is measured on a number of real-life C++ benchmarks. Although some of these benchmarks are not typical examples of embedded applications, we believe they represent a broad range of the code properties of the targeted (embedded) applications: applications written in C++ using more or less library code and using more or less templates and inheritance. Some properties of the benchmark programs are summarized in Table 1: a short functional description is provided, together with the number of assembly instructions in the base versions of the applications. It are these base versions with which the compacted versions will be compared. They were generated using three tools:

1. Compaq's C++ V6.3-002 compiler was used to compile with the -O1 -arch ev67 flags, resulting in the smallest binaries the compiler can generate.
2. The Compaq Linker for Tru64Unix 5.1 (the operating system used for our evaluation) was used to link the programs with the flags -Wl,-m -Wl,-r -Wl,-z -Wl,-r -non_shared, resulting in statically[3] linked binaries including the relocation and symbol information necessary for SQUEEZE++, and resulting in a map of the original object file code and data sections in the final program. SQUEEZE++ requires this map for its analyses discussed in [7].
3. A very basic post link-time program compactor. It eliminates no-ops from the programs and performs an initial elimination of unreachable code. It applies no-op elimination on the linked binaries because a code-size-oriented compiler would not have inserted no-ops in the first place, and an initial unreachable code elimination because this compensates for the fact that the Compaq system libraries are not really structured with compact programs in mind: the object files in the libraries have a larger granularity than one would expect from a library oriented at compact programs, as more coarse-grain object files generally result in more redundant code being linked with a program. It also applies the same code scheduling and layouting algorithms as SQUEEZE++, using the same profiling information for the benchmarks. Using the same scheduler and layouter allows us to make a fair comparison between the execution speeds of different versions of the benchmarks.

As the prototype SQUEEZE++ only handles statically linked programs, including both application-specific and library code, the fraction of the program code that we consider application-specific is also given. All system libraries (libc, libcxx, libstdcxx, libX11, etc.) are considered library code and not application-specific. Libraries that typically are used by more applications on a system are also not considered application-specific. For the lyx benchmarks, e.g., the forms GUI-library is not considered application-specific, while the xforms library is considered application-specific, as it is written as a wrapper around the forms library to ease the implementation of lyx, and only of lyx. Although we have linked xforms into the program as a library, such that the linker only includes the necessary parts of the xforms library (this part may depend, e.g., on the target platform), we still consider it part of the application-specific code. A similar reasoning holds for the GTL library: it is a library, so we link it with the program as a library. Still we consider it to be application-specific, as it represents the kind of libraries that is typically not used in multiple applications on a system.

The table also includes the fraction of the application-specific code that comes from the so called repository, i.e. the fraction of the code that was generated for template instantiations. As discussed in section 6, a repository is used to avoid the inclusion of multiple identical template instantiations or specializations into a program. We have included these fractions, because they give an idea of the size of the code that we explicitly target with our techniques for whole-procedure reuse and parameterization.

5.2 Whole-Program Code Compaction

The code abstraction performance of SQUEEZE++ is evaluated by comparing three versions of the benchmark programs: the base version for our comparison is the version generated as described in the previous section. The second version of the programs are the compacted binaries, but in which no code is reused: i.e. the programs are compacted applying all the available techniques in SQUEEZE++ (including constant propagation, useless code elimination, unreachable code elimination, etc., see [7, 10] for an extended discussion) except the techniques discussed in this paper. The third version is generated by SQUEEZE++ applying all available compaction and code reuse techniques.

The results for our benchmark programs are depicted in Figure 5. The lower line indicates the compaction achieved by SQUEEZE++ without the code abstraction techniques. As can be seen, 26%-39% of the code is eliminated, averaging around 31%.

If code abstraction and parameterization are applied as well, the average code size reduction climbs to 45%. This reduction fluctuates between 34 and 62%. The additional code size reduction resulting from the code reuse thus averages around 14%, ranging from 6% to 24%.

5.3 Application-specific code

One can question the usefulness of measuring code size reductions on statically linked programs. Aren't smaller programs among the benefits of using shared libraries? Our answer to this is twofold.

First of all, we think statically linked programs are still very often used on embedded applications where code size matters. A typical example is the implementation of command line tools, such as for embedded Linux systems. These include tools such as ls, cat, more, less, echo, chown, chmod,

[3]Unless programs are statically linked, the Compaq linker will not include the relocation information needed by SQUEEZE++ in the generated binary.

Program	Description	# instr.	application-specific code	repository code
blackbox	Fully functional, lightweight window manager	328240	14%	0%
bochs	Virtual Pentium machine	275440	37%	0%
lcom	"L" hardware description language compiler	99168	31%	0%
xkobo	Arcade space shooter game	252800	4%	0%
fpt	In-house HPF automatic parallelization tool	530896	32%	7%
252.eon	Probabilistic ray tracer (from the SPECint2000 suite)	136224	51%	10%
lyx	WYSIWYM(ean) word processor (LaTeX-like documents)	1329616	66%	23%
gtl	Test program from the Graph Template Library (GTL)	161936	60%	47%

Table 1: Description and properties of the programs on which our code compaction was evaluated.

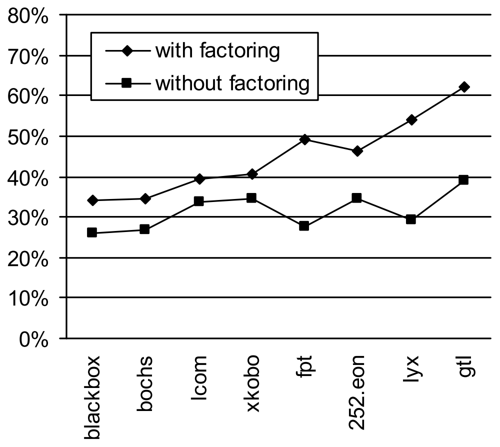


Figure 5: Code size reductions achieved on statically linked program.

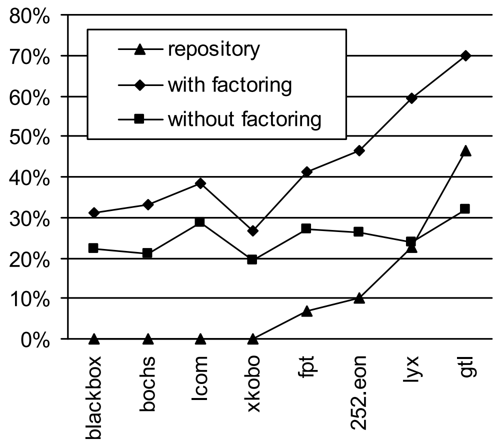


Figure 6: Code size reductions achieved on application-specific part of the program.

mkdir, etc. Instead of implementing these tools with multiple applications and shared libraries, only one statically linked program is generated. This single program includes the functionality of all separate tools. It is installed in the /bin directory and symbolic links to it are created, each link corresponding to one of the original command-line tools. The program evaluates the command-line command (i.e. the name fed by the user at the command-line) and the requested functionality is performed. By using a single statically linked program, only those parts of the libraries needed for all provided functionality are linked with the program, thus avoiding the overhead of dynamic linking. This or similar techniques are also used for implementing multiple applications on embedded systems.

The second and more interesting part of our answer is the evaluation of our techniques on the application-specific code of the program, i.e. the code that also is found in the dynamically linked programs. To approximate the performance of a Squeeze++-like tool for dynamically linked applications as well as possible, we have adapted Squeeze++ in two ways:

1. At call-sites in application-specific code where library code is called, we have not resolved the calls. This means that at the call-site, we do not know the callee. We only know that the callee respects the calling-conventions. Vice versa, the thus called library code is called from unknown calling contexts, assuming only that their callers respect the calling conventions as well. This is exactly the same as what compilers do when callees are externally declared or when procedures are exported.
2. For code reuse, all fingerprints used for prepartitioning include a tag indicating whether the code belongs to the application-specific or the library part of the program. Identical, similar or equivalent code fragments are therefore only detected within either application-specific or library code and the code reuse is completely separated for both parts.

With this adaptation, application-specific code is completely separated from library code in Squeeze++, and the code reuse techniques applied to the application-specific code are exactly the ones that are also applicable on dynamically linked programs. The only remaining difference with dynamically linked programs is that there are no stubs for implementing the calls to shared library code. The influence of these stubs on the results from applying our code reuse techniques are in our opinion neglectible.

Figure 6 shows the same information as Figure 5, but now for application-specific code only. As can be expected, the performance of the general optimization and compaction techniques is lower on the application-specific code. The reason is precisely that this code is application-specific and therefore fewer possibilities are found to optimize the code for the application. By and large the general optimization and compaction possibilities for application-specific code result from the inefficiencies of separate compilation. From library code, by contrast, much more code is unreachable in some application because it is redundant for that specific application. The average application-specific code size reduction due to general optimization and compaction techniques is 25% (compared to 32% for the whole programs). This reduction ranges from 19 to 32% (compared to 26 to 39% for the whole programs).

At least for some of the benchmarks, the code abstraction and parameterization techniques perform much better on the application-specific code, resulting in a total average code size reduction of 43%. This is still lower than the average reduction achieved on whole programs (45%). For some programs however, the maximal achieved compaction on application-specific code is larger than the compaction on the whole program. The maximum compaction now ranges from 27 to 70% (compared to 34 to 62% for whole programs).

For some programs, the code reuse techniques perform much better on application-specific code than on the whole program, as can be seen when comparing Figures 5 and 6. The average additional application-specific code size reduction due to code reuse techniques now ranges from 7 to 38% (compared to 6 to 24% for the whole programs), averaging around 18% (compared to 14%). This stronger performance of the code reuse techniques on application-specific code can be explained by looking at the third line we've included in Figure 6, indicating the fraction of application-specific code that comes from the repository. There is a very strong correlation between the performance of the code reuse techniques and the amount of code originating from templates. This corresponds to the claims we have made about reusable code generated for template specializations, even when repositories are used to avoid to some extent the linking of duplicate code fragments.

5.4 Compaction Times

The common knowledge about whole-program analyses and optimizations is that they are notoriously slow and therefore often not practical. In Figure 7, we have depicted the time needed to fully compact the binaries with SQUEEZE++, i.e. to apply all the techniques at our disposal. We believe these compaction times show that link-time optimizations are practical, as there is no need to apply them during each edit/compile/debug cycle.

It is important to note that SQUEEZE++ is a research prototype. We have only optimized it to facilitate our research and to shorten our edit/compile/debug cycles. So while we, e.g., have optimized the liveness analysis, it is still fully separated from all other implemented techniques. Before any of the other techniques that rely on liveness information are applied, liveness analysis is restarted from scratch. It would be much more efficient if all or at least some of the program transformations would update the liveness information on the fly. We have opted not to do so, because it would severely limit the ease with which SQUEEZE++ can be adapted to try out new techniques.

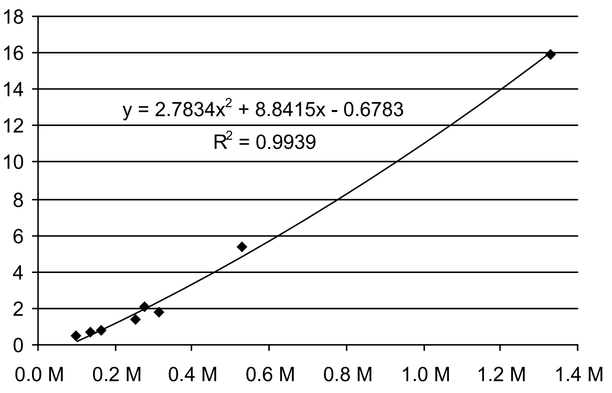


Figure 7: Compaction times (in minutes) in function of the number of instructions in the binaries.

5.5 Breakdown of the Results

In the lower graph of Figure 8 we show the contributions of the different code reuse techniques to the total additional code size reduction achieved by these techniques combined. These contributions are additive, since more fine-grained techniques are applied where coarse-grained techniques failed to find multiple-occuring code fragments. As we consider the numbers for the application-specific code to be more relevant, we have depicted the graph for these numbers. It is clear from the graph that whole-procedure reuse and abstraction perform best where templates are most used. It is also clear that the importance of the more fine-grained techniques lowers as more templates are used. Also note that, as the lines in the graph for procedure parameterization and code region abstraction are basically the same, code region abstraction has lost most if not all of its merits.

5.6 Influence on Compaction Speed

In the upper graph of Figure 8, we have depicted the (additive) slowdowns of SQUEEZE++ when code reuse techniques are applied. Several things can be said about the results shown in this graph. First of all notice that whole-procedure reuse and parameterization seem almost free. This is not because these techniques consume few cycles compared to the other optimization techniques. The reason is that these techniques are applied relatively early in the compaction process. SQUEEZE++ consists of a number of phases:

1. disassembly and CFG construction;
2. trivial optimizations (basic version);
3. iterative optimizations;
4. optimizations applied only once;
5. iterative optimizations;
6. fine-grained code reuse;
7. iterative optimizations;
8. code layout, scheduling and assembling.

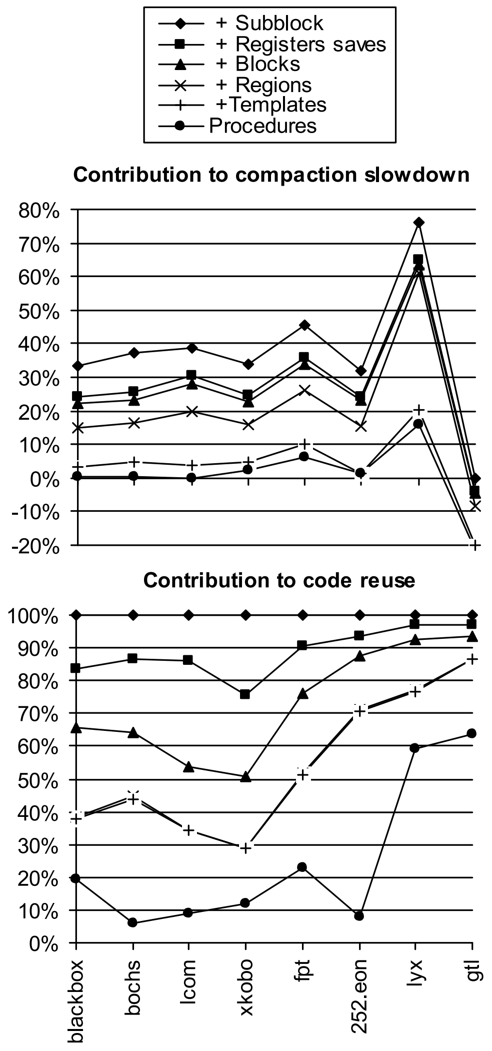


Figure 8: Contributions to code size reduction and compaction slowdown for the various code reuse techniques.

The iterative optimizations include constant propagation, unreachable and dead code elimination, CFG refinements, etc. The optimizations that are applied only once include some CFG refinements and the inlining of procedures with only one calling context.

As whole-procedure code reuse targets identical procedures, we do not want to take the risk that differences are introduced in initially identical procedures by the application of the iterative optimizations. On the other hand, these optimizations might remove differences between procedures. To exploit both possibilities, data reuse and whole-procedure reuse are applied at the beginning of each run over the iterative optimizations. As this immediately eliminates a large amount of code for some of the benchmarks, all the later techniques in Squeeze++ are applied on a smaller program, hence the overall speedup on some benchmarks due to the reuse of whole procedures.

As parameterization targets similar whole procedures, it is of no use for nearly identical procedures that have been inlined in different contexts. Once they are inlined, we would not consider them as procedures any more. On the other hand, as parametrization involves the introduction of some overhead, we want to avoid parametrizing two procedures of which we later could find out that one of them was unreachable. Therefore procedure parameterization is applied just prior to the inlining of procedures with only one calling context. At that time, most unreachable code that we can detect has been detected and removed. Again, the algorithms in later phases are applied on a smaller program.

Two further remarks should be made: it looks as if (the almost useless) code region abstraction is responsible for the largest slowdown. A large part of the computations needed for abstracting code regions however are also needed for basic block abstraction. Therefore, the relative weight in the slowdown because of code region abstraction is overestimated in the graph. Finally, it is important to know that these numbers were collected using the Squeeze++ version that keeps application-specific code and library code separated. All techniques are applied on both of them, albeit separated. The slowdowns measured therefore are an overestimation of the actual slowdowns when the techniques would have been applied on application-specific code only. This is because most algorithms scale superlinearly and because the whole-procedure reuse and parameterization techniques do not at all reduce the size of the library code to be handled by later phases in Squeeze++ like they do on application-specific code.[4]

5.7 Influence on Execution Speed

As code reuse for minimizing the static code size is applied, control flow transfers and copy instructions are inserted in the program. Therefore the dynamic number of executed instructions increases with code reuse and we can expect that execution times increase as well with code reuse. On the other hand, code reuse can improve cache behavior and have a positive influence on execution speed. To measure the combined positive and negative influence on execution speed, we've measured the execution times of 4 versions of some benchmark programs. These versions are:

- The base binaries.
- The compacted binaries where the reuse of identical procedures is the only applied code reuse technique, since it is the only reuse technique that introduces no overhead (no overhead reuse).
- The fully compacted binaries (all reuse).
- A version of the programs, where all code reuse techniques are applied, but the ones that involve overhead are only applied on cold code, i.e. code that according to the profile information is not frequently executed. We took as cold code that part of the code with the lowest execution frequencies, that is responsible for the bottom 5% of the number of dynamically executed instructions (cold code reuse).

[4]Because of dependencies between the different general compaction techniques, major parts of Squeeze++ would require an (infeasible) rewrite not to apply the general compaction techniques on the library code. As we have not rewritten Squeeze++ for this purpose, not applying the abstraction techniques to the library code would have resulted in an underestimation. We prefer presenting overestimated slowdowns.

The latter three versions were generated with (adapted) versions of SQUEEZE++ that separate application-specific code from library code. Figure 9 shows the code size reductions for the application-specific part of the three compacted versions of the programs. It is clear that the maximal reductions are very well approximated when reuse techniques are applied to cold code only.

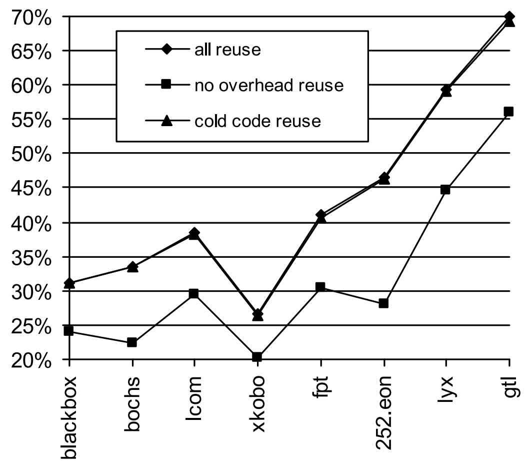


Figure 9: Code size reductions for the three compacted versions of the program.

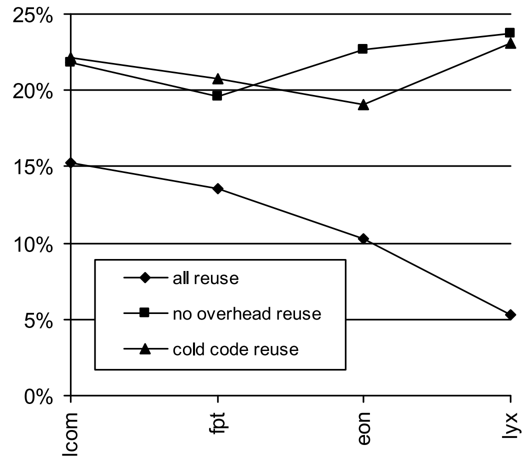


Figure 10: Speedups for the three compacted versions of the program.

In Figure 10, the speedups compared with the base versions are shown. The experiments were run on a 667 MHz Compaq Alpha 21264 EV67 processor with a split primary cache (64 KB each of instruction and data cache), 8 MB of off-chip secondary cache, and 1.5 Gbytes of main memory running Tru64 Unix 5.1. We were able to generate reliable timing results for 4 of the 8 benchmarks. The other programs are either highly interactive, provide no batch mode for execution, or we had no input data available that lead to high enough execution times for accurate measurements. For these four programs, the input data we used for measuring the speedups are significantly larger and different from the input data used for training the programs and generating the profiles.

As can be seen from the graph, the speedups obtained by all the optimizations in SQUEEZE++ and the reuse techniques involving no overhead are significant. Blindly applying all the code reuse techniques results in a much lower speedup, and as more code is reused (left to right on the graph), the obtained speedup shrinks. Selectively applying the reuse techniques to cold code however does not result in a slowdown (or smaller speedup). Sometimes, because of improved cache behavior, the application even runs faster when overhead is introduced in abstracted cold code.

Note that these timing results are indicative only. Because of the very strict scheduling and code layout rules to be able to issue multiple instructions in one cycle, the execution times of different versions on our superscalar processor are highly sensitive to very small variations in the layout and the schedule of the generated code. Since no no-ops are inserted during scheduling to optimize the schedule and code layout, a fraction of the timing results depends on pure luck. Given the fact that our timings results show consistent trends for all four benchmarks, we however feel confident about our conclusions: by selectively applying code reuse techniques, the maximum possible code compaction can be approximated very well, without sacrificing execution speed.

5.8 Discussion

One can ask whether the discussed techniques will perform better at compile-time or at link-time. We feel that they will perform better at compile-time, where more semantic information is available that can be used (1) to overcome the stack frame constraints we encountered, and (2) to build more precise call graphs. The latter is particularly important for the whole-program optimizations we apply (but which are not discussed in this paper). On the other hand a large part of the code size reduction we obtain is the direct or indirect result of knowing the locations at which statically allocated data is stored in the final program. These locations are determined by the linker. So if one moves the related optimizations into the compiler, the linker functionallity needs to be moved into the compiler as well. Compile-time then equals link-time.

Being able to perform the techniques at link-time, i.e. after the semantical information is lost, also allows to deal to a large extent with code size constraints without changing some principles of today's C++-like programming environments, such as separate compilation and libraries being available in object format only. We feel that, before dropping these principles and moving to whole-program compilation because of their disadvantages, the possible strenghts and weaknesses of link-time compaction should be well understood.

6. RELATED WORK

6.1 Code Compression

There is a considerable body of work on code compression, but much of this focuses on compressing executable files as much as possible in order to reduce storage or transmission costs. These approaches generally produce compressed representables that are smaller than those obtained using our approach, but have the drawback that they must either be decompressed to their original size before they can be executed [12, 14, 15, 16, 27]—which can be problematic for limited-memory devices—or require special hardware support for executing the compressed code directly [21, 22, 23, 35, 36]. By contrast, programs compacted using our techniques can be executed directly without any decompression or special hardware support.

6.2 Code Abstraction

Most of the previous work on code abstraction to yield smaller executables treats an executable program as a simple linear sequence of instructions [2, 6, 17]. They use suffix trees to identify repeated instructions in the program and abstract them into procedures. The size reductions they report are modest, averaging about 4–7%. Clausen et al. [4] applied minor modifications to the Java Virtual Machine to allow it to decode macros that combine frequently recurring bytecode instruction sequences. They report code size reductions of 15% on average. Our techniques do not rely on changing the underlying architecture on which a program is executed and are not language dependent.

Fraser and Proebsting [18] look for repeated patterns in the intermediate program representation used by the compiler. So called super-operators are chosen, corresponding to the most frequently occurring patterns. These (application-specific) super-operators are used to extend a virtual instruction architecture, for which the program is compiled. At the same time, an interpreter that is able to interpret the extended instruction set is generated in C, from which it can be compiled to the original target architecture. They report an average code size reduction of 50%, albeit with an undesirable large impact on execution speed.

The techniques discussed in this paper are fully language independent, do not require any modifications to the compilers or the target architecture and produce programs that are faster, rather than slower.

6.3 Other Program Compaction Techniques

The elimination of unused data from a program has been considered by Srivastava and Wall [31] and Sweeney and Tip [32]. Srivastava and Wall, describing a link-time optimization technique for improving the code for subroutine calls in Alpha executables, observe that the optimization allows the elimination of most of the global address table entries in the executables. However, their focus is primarily on improving execution speed, and they do not investigate the elimination of data areas other than the global address table. With our previous work [7], the same optimizations are applied, but in a more general way and not limited to the global address table.

Sweeney and Tip [32] focus on the removal of dead data members from classes in C++ programs. They report a run-time high watermark (i.e. the largest object space needed during the execution of the program) reduction of 4.4% on the average. This is the result of the elimination of 12% of the data members.

For object-oriented programming languages, several techniques have been proposed for application extraction, where only the necessary parts of libraries and/or run-time environments are linked with the programmer's code. For Self [1], a dynamically typed language, such systems obtain higher compaction levels than our system. They are however to some extent programming-language specific and start from programs containing the whole run-time environment of Self applications.

Tip et al. [33] achieve results for Java programs that are very similar to our results. Although most of their techniques are based on language-independent algorithms, e.g., for building a call graph of a program [34], some of the applied optimizations are language dependent, such as the compaction of the constant pools in Java programs. Besides that, their techniques exploit the type information that is available in the Java bytecode. We do not use such information, as it is not or hardly available in native binary programs. Srivastava has studied the removal of unreachable procedures in object-oriented programming environments [30].

MLD [13] and Vortex [8] are two whole-program optimizers for object-oriented languages (that are not specifically aimed at code compaction however). They focus on reducing the overhead created by virtual method invocation. Looking at the whole class hierarchy of a program, some of the virtual method invocations can be replaced by direct ones. These systems also reduce the performance penalty due to polymorphism by using profile information to optimize the method calls for the most frequently appearing object types.

6.4 Binary Rewriting and Binary Translation

Static binary rewriting at link-time somehow seems very appropriate for the Alpha architecture. Several static link-time binary optimizers have been developed for Alpha powered systems. These include OM [31], Spike [5] and Alto [26], that focus on speed optimization. Alto is the most advanced of them, and a binary optimizer quite similar to Alto has been implemented for the IA-32 architecture: PLTO [29].

We know of two static binary rewriting tools for embedded systems targeting code size. CodeCompressor [39] from Raisonance is a code compactor applying inlining, code abstraction (we have found no details on their techniques) and peephole optimizations on programs compiled for the 8051 architecture. The creators of this commercial tool expect program size reductions of up to 25%. aiPop [38] is a more sophisticated post link-time code compactor for the C16x architectures. It includes, e.g., constant propagation, peephole optimizations, code abstraction, procedure tail merging and dead code elimination. The reported code size reductions range from 4 to 20%. Besides the techniques discussed in this paper, SQUEEZE++ implements a broad range of whole-program analyses and optimizations. These include a.o. peephole optimization, copy propagation, load/store avoidance, constant propagation, dead code elimination, unreachable code elimination, dead data elimination, inlining and code layout optimizations. We refer to [10, 7] for a more detailed discussion.

There has been a great deal of interest in dynamic binary optimization (see, for example, [40, 20, 37]). In these approaches, however, the data structures necessary for run-time execution monitoring and optimization incur nontrivial additional memory overheads, and hence are not suited for the goal of this work, which is memory footprint reduction of applications. For this reason, we do not discuss them further.

6.5 OOPL-Specific Linking Mechanisms

If identical template instantiations occur in multiple modules from a program, several techniques [25] can be used to avoid that these instantiations are linked with the program multiple times.

One of these techniques is incremental linking, where the compiler initially generates no instantiations at all. The linker notices that some code and or data cannot be retrieved and feeds this back to the compiler, who generates the necessary instantiations. This technique only provides a way to avoid the linking of multiple identical instantiations at the source code level, as it is based on the names of the symbols the linker does not find. These names include, through name mangling, the types of the objects for which the templates were instantiated. Even when long and int are identical on some machine, this scheme will not avoid linking both Stack<long> and Stack<int> instantiations with a program.

Another approach is the use of a so called repository, a database consisting of all the instantiated templates. As in relational databases, all records are unique, thus avoiding multiple identical instances of the same template. As the records in the repository are identified by names, these repositories have the same limitations as incremental linking. Some Microsoft linkers however are able to compare the code in the records in the repository. [25].

A third technique is used by the GNU compilers: all sections in object files that were generated for instantiating templates have a special tag: .gnu.linkonce.d. The linker compares these sections (again using symbol names only) and thus avoids multiple occurrences of the same template instantiation in the final program.

It is clear that these techniques do not address the occurrence of code fragments like SQUEEZE++ does.

Very different techniques to avoid code growth because of using template-like language features have extensively been researched in the past, especially for the Ada programming language and its so called generics. Most of the proposed techniques use polymorphism [3, 28] to avoid the need for static specialization of the generics used. The consequence of those techniques is that no optimized specializations are generated, but to the contrary, overhead is introduced to implement the polymorphism. Furthermore, they do not reuse code at the (sub) basic block level.

7. CONCLUSIONS

Generally applicable program compaction, applied at link-time, is able to achieve significant code size reductions for applications developed in object-oriented programming languages such as C++. The main opportunities come from the use of reusable code: libraries developed with code reuse and general applicability in mind on the higher level and programming constructs such as templates and inheritance on the lower level. The code reuse techniques for reusing whole procedures introduced in this paper contribute significantly to the achieved results.

Our prototype link-time code compactor, named SQUEEZE++, achieves average code size reductions of 45% on a set of 8 real-life statically linked C++ applications, ranging from 34 to 62%, without sacrificing execution speed. On the application-specific part of the programs, which is representative for dynamically linked programs, even better results are obtained when language features such as templates are used by the program. These code size reductions do not lead to slower programs. In contrast, when the code reuse techniques are applied selectively, the speedups obtained by the general code optimization techniques implemented in SQUEEZE++ can still be obtained.

Acknowledgements

The work of De Sutter was supported by the Fund for Scientific Research – Flanders under grant 3G001998. De Bus is supported by a grant from the 'Flemisch Institute for the Promotion of the Scientific Technological Research in the Industry' (IWT). We would like to thank Saumya Debray and the anonymous reviewers for their very useful remarks and suggestions on previous version of this paper.

8. REFERENCES

[1] O. Agesen and D. Ungar. Sifting out the gold: Delivering compact applications from an exploratory object-oriented environment. In *Proc. 1994 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 355–370, Oct. 1994.

[2] B. S. Baker and U. Manber. Deducing similarities in Java sources from bytecodes. In *USENIX Annual Technical Conference*, pages 179–190, June 1998.

[3] G. Bray. Sharing code among instances of Ada generics. In *Proc. 1984 ACM SIGPLAN Symposium on Compiler Construction (CC)*, pages 276–284, June 1984.

[4] L. Clausen, U. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, May 2000.

[5] R. Cohn, D. Goodwin, P. Lowney, and N. Rubin. Spike: An optimizer for alpha/nt executables. In *USENIX Windows NT Workshop*, Aug. 1997.

[6] K. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 139–149, May 1999.

[7] B. De Sutter, B. De Bus, K. De Bosschere, and S. Debray. Combining global code and data compaction. In *Proc. 2001 ACM SIGPLAN Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 29–38, June 2001.

[8] J. Dean, G. DeFouw, D. Grove, V. Litvinov, and G. Chaimber. Vortex: an optimizing compiler for object-oriented languages. In *Proc. 1996 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 83–100, Oct. 1996.

[9] S. Debray and W. Evans. Profile-guided code compression. In *Proc. 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 95–105, June 2002.

[10] S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming Languages and Systems*, 22(2):378–415, Mar. 2000.

[11] Embedded C++ Technical Committee. *The Embedded C++ Specification*, Oct. 1999.

[12] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting. Code compression. In *Proc. 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 358–365, June 1997.

[13] M. Fernández. *A Retargetable, Optimizing Linker*. PhD thesis, Princeton University, Jan. 1996.

[14] M. Franz. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile-object systems. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, number 1222 in LNCS, pages 263–276. Springer, Feb. 1997.

[15] M. Franz and T. Kistler. Slim binaries. *CACM*, 40(12):87–94, Dec. 1997.

[16] C. Fraser. Automatic inference of models for statistical code compression. In *Proc. 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 242–246, May 1999.

[17] C. Fraser, E. Myers, and A. Wendt. Analyzing and compressing assembly code. In *Proc. 1984 ACM SIGPLAN Symposium on Compiler Construction (CC)*, pages 117–121, June 1984.

[18] C. Fraser and T. Proebsting. Custom instruction sets for code compression. http://research.microsoft.com/˜toddpro, 1995.

[19] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel. A code compression system based on pipelined interpreters. *Software Practice and Experience*, 29(11):1005–1023, 1999.

[20] R. Hookway and M. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1):3–12, 1997.

[21] T. M. Kemp, R. M. Montoye, J. D. Harper, J. D. Palmer, and D. J. Auerbach. A decompression core for powerpc. *(IBM) J. Research and Development*, 42(6), Nov. 1998.

[22] D. Kirovski, J. Kin, and W. H. Mangione-Smith. Procedure based program compression. In *MICRO*, Dec. 1997.

[23] K. D. Kissell. MIPS16: High-density MIPS for the embedded market. In *Proc. of Real Time Systems '97 (RTS97)*, 1997.

[24] C. Lefurgy. *Efficient Execution of Compressed Programs*. PhD thesis, University of Michigan, June 2000.

[25] J. Levine. *Linkers & Loaders*. Morgan Kaufmann Publishers, 2000.

[26] R. Muth, S. Debray, S. Watterson, and K. De Bosschere. alto : A link-time optimizer for the Compaq Alpha. *Software Practice and Experience*, 31(1):67–101, Jan. 2001.

[27] W. Pugh. Compressing java class files. In *Proc. 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 247–258, May 1999.

[28] J. Rosenberg. *Generating Compact Code for Generic Subprograms*. PhD thesis, Carnegie-Mellon University, 1983.

[29] B. Schwarz, G. Andrews, M. Legendre, and S. Debray. PLTO: A link-time optimizer for the intel ia-32 architecture. In *Proc. Workshop on Binary Translation (WBT)*, Sept. 2001.

[30] A. Srivastava. Unreachable procedures in object-oriented programming. *ACM Letters on Programming Languages and Systems*, 1(4):355–364, Dec. 1992.

[31] A. Srivastava and W. Wall. Link-time optimization of address calculation on a 64-bit architecture. In *Proc. 1994 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 49–60, June 1994.

[32] P. Sweeney and F. Tip. A study of dead data members in C++ applications. In *Proc. 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 324–323, June 1998.

[33] F. Tip, C. Laffra, and P. Sweeney. Practical experience with an application extractor for java. In *Proc. 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 292–305, Nov. 1999.

[34] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *Proc. 2000 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 281–293, Oct. 2000.

[35] TriMedia Technologies Inc. *TriMedia32 Architecture*, Oct. 2000.

[36] J. Turley. Thumb squeezes ARM code size. *Microprocessor Report*, 9(4):1–5, Mar. 1995.

[37] B. Vasanth, E. Duesterwald, and S. Banejia. Dynamo: A transparent dynamic optimization system. In *Proc. 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–12, June 2000.

[38] http://www.absint.com/aipop/.

[39] http://www.raisonance.com.

[40] http://www.transitives.com.