Combining Global Code and Data Compaction*

Bjorn De Sutter
Bruno De Bus
Koen De Bosschere
Ghent University, Belgium
brdsutte@elis.rug.ac.be

Saumya Debray
The University Of Arizona
debray@cs.arizona.edu

ABSTRACT

Computers are increasingly being incorporated in devices with a limited amount of available memory. As a result research is increasingly focusing on the automated reduction of program size. Existing literature focuses on either data or code compaction or on highly language dependent techniques. This paper shows how combined code and data compaction can be achieved using a link-time code compaction system that reasons about the use of both code and data addresses. The analyses proposed rely only on fundamental properties of linked code and are therefore generally applicable. The combined code and data compaction is implemented in SQUEEZE, a link-time program compaction system, and evaluated on SPEC2000, MediaBench and C++ programs, resulting in total binary program size reductions of 23.6%–46.6%. This compaction involves no speed tradeoff, as the compacted programs are on average about 8% faster.

1. INTRODUCTION

Computers are increasingly 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, there is a desire to execute increasingly sophisticated applications, such as encryption and speech recognition, on such devices. This leads to increasingly large programs, due to the additional functionality they provide, as well as the use of modern software engineering techniques that aim at the use of components or code libraries. 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, and because of the complex structure of these building blocks, the linker often links a lot of useless code and data into the application. This problem can be considered as one of the big hurdles to be taken before modern software engineering techniques can be used to develop mobile or embedded applications.

For these reasons, recent years have seen growing interest in research on code and data compaction, i.e., the transformation of programs to reduce their memory footprint while keeping them directly executable. Work on code compaction has generally focused on identifying repeated instruction sequences within a program and abstracting them into functions [6, 14] or macro-instructions in programmable execution environments such as the Java Virtual Machine [4]. Work on data compaction is limited to simple literal address removal from object files [21]. Whereas program compaction compacts code and data in a program, program extraction identifies those parts of libraries, classes or run-time environments that are not needed for a specific application. To our knowledge, such proposed techniques [1, 22, 23] are language dependent, requiring higher level descriptions of libraries, classes or run-time environments and above all type information. This highly limits their applicability, e.g., on libraries that are available in object format only.

In the past we have proposed applying code compaction on a very general program representation: binary programs. The techniques discussed were limited to code compaction only. However the elimination of a word of storage from the data area of a program yields exactly the same overall benefit, in terms of memory footprint reduction, as the elimination of a word of storage from the code area of the program. Moreover, it is not difficult to see that there are significant dependences between the code and data components of an executable program. For example, unused library code that is uselessly being linked with a program will often be accompanied by useless data (empirical evidence indicates that 5–10% of the library code linked with a program is unreachable [18, 20]). Code optimizations such as dead and unreachable code elimination can cause data to become unreachable by getting rid of code referring to that data. Conversely, the elimination of unused data that contains pointers to code, such as jump tables and virtual function tables, can cause code to become unreachable, and hence eliminable. Indeed, the two optimizations are synergistic: the elimination of data can enable additional elimination of code, which can enable the elimination of even more data, and so on.

The main contribution of this paper is to develop a whole-program analysis that treats data and code elimination from binary programs simultaneously. We show how this can be done using a link-time code compaction system that reasons about both code and data addresses. Conceptually, the idea is very simple: use constant propagation to determine the values of addresses in code and data areas, and based on this reasoning identify code and data values that are not used and can be eliminated. The link-time program compaction system SQUEEZE, in which our new algorithms were imple-

*The work of B. De Sutter was supported by the Fund for Scientific Research – Flanders under grant 3G001998. B. De Bus's is supported by a grant from the 'Flemisch Institute for the Promotion of the Scientific Technological Research in the Industry (IWT). The work of S. Debray was supported in part by the National Science Foundation under grants CCR-0073394, EIA-0080123, and ASC-9720738.

LCTES 2001, Snowbird, Utah, USA

main.c:

```
#include <stdio.h>
main() {
  printf("hello world\n");
}
```

pointer.c: /* contains dead code only */

```
#include <stdio.h>
void a(void) {
  printf("%lx\n",(void*)&a);
}
```

(a) Source files

Binary	main.o pointer.o		main.o	
	data	code	data	code
original	70848	182432	46704	58432
code compacted	68032	102016	45680	24640
data compacted	43408	20864	43408	20864

(b) Executable sizes in bytes

Figure 1: A motivating example

mented and evaluated, achieves size reductions that are significantly better than have been reported in the past: we achieve reductions of about 24.4%–50.7% in the code size, and 23.6%–46.6% in the total program size (code+data). Our ideas rely only on general properties of compiled code and so are not restricted to a particular implementation context. For simplicity of exposition the discussion below will focus on load-store Harvard architectures, where arithmetic operations involve only registers, and memory is accessed only via `load` and `store` instructions. However, the ideas presented here are not limited to such architectures, and can be readily adapted to architectures supporting more complex addressing modes.

2. MOTIVATING EXAMPLE

During a linking process, the search for necessary library code and data to be included in the binary is guided by symbol information. If a symbol is referred to in some already linked-in object file, another object file defining that symbol will be linked with the binary as well, possibly requiring new symbol definitions. This is an iterative process, that finishes when all referenced symbols are defined. Whether the reference to a symbol is going to be used by the program is not taken into account during this process.

Consider the small example C-program in Figure 1(a), which consists of two source code files, one of which contains dead code only. Depending on whether or not the object file containing the dead code is fed to the linker together with the main program, different binaries are generated, whose data section sizes and code section sizes are given in Figure 1(b). The addition of the dead code object file results in a binary that has more than twice the amount of data and code, although the two object files themselves are only 1KB large. The sole reason for this is the additional inclusion of object files from the C-library caused by the call to `printf()`, in the dead code, to output an integer number.

When code compaction is applied to both the binaries, a significant amount of code is eliminated from them. Still one version contains much more code than the other. The reason is that the dead procedure `a()` cannot be eliminated, since its address is stored in the data (for printing it). Unless we are able to analyze all the possible uses of this stored address, we must assume that it can be used as a procedure pointer and that as a result `a()` is a potential target of indirect procedure calls.

When the analyses proposed in this paper are applied, the resulting binaries are equal in size and contain the same (smaller) amounts of code and data.

While this example is admittedly contrived, the point it is intended to illustrate is that even a very small amount of unreachable code, with only a small amount of additional functionality—in the example shown, a request to print an integer—can have a nontrivial effect on code size. In real programs, it often happens that such unreachable parts of library object files are linked with the program, iteratively causing other code and data to be linked with it as well, resulting in a significant overall impact on program size.

3. STRUCTURE OF COMPILED CODE

The object module generated by a compiler from a source module typically consists of several sections, such as the text section, the constant data section, the literal address section, etc. The linker combines a number of such object modules into an executable program: in the process, it puts all the sections in their final order and location. The sections of the same type coming from different object modules are typically combined into a single section of that type in the final executable. To avoid confusion, in the remainder of this paper the original sections in the object files will be called code and data blocks, or blocks for short. A section in an executable file is thus a juxtaposition of blocks from the object modules from which the executable was constructed.

To access a memory location, the address of that location has to be loaded or computed into a register (possibly implicitly, as a displacement off a base address). In general, when generating the blocks in one object module, the compiler does not have any information about the blocks in other object modules, such as their size or the order in which they will be linked together, so it cannot make any assumptions about the eventual locations of these blocks in the final executable. This means that in the object code, computations on an address pointing to some block can never yield an address pointing to some other block in the object file, because the displacement between the two blocks is not known at compile time. This property holds for all the blocks in the final executable program. This means that the data in a block is dead unless there is a pointer to that block found in some other block (e.g., a pointer to a data block from a code block, or vice versa) or explicitly programmed in the code.[1] If there are such pointers, but they are not used for

[1]It is possible, in principle, for a program to communicate such pointers from one point in a program to another in nonstandard ways, e.g., by writing it out to a file at one program point and reading it back in at another. The discussion here applies even in such situations. For example, in order to write out an address, we have to first put the address into a register, so we can detect that the address is taken; at the other end, code that attempts to dereference a value that

non-zero initialized data	2552912 bytes
non-zero intialized read-only data	1115392 bytes
relocatable data	473580 bytes
read-only relocatable data	405412 bytes
block size = 8 bytes	32184 blocks
block size = 16 bytes	3603 blocks
16 bytes < block size ≤ 64 bytes	738 blocks
64 bytes < block size ≤ 256 bytes	882 blocks
256 bytes < block size ≤ 1KB	487 blocks
1KB < block size ≤ 4KB	257 blocks
4KB < block size ≤ 16KB	116 blocks
16KB < block size ≤ 64KB	22 blocks

Table 1: Some numbers on statically allocated non-zero-initialized data and addresses summed for the whole SPECint2000 benchmark suite.

stores, the data is read-only.

This property is fundamental to the analyses described later in this paper, in Sections 4 and 5. Both analyses are extensions to constant propagation that, based on the propagated addresses and how these addresses are used, are able to detect dead and read-only memory areas, and each algorithm has its strengths and weaknesses. In Section 6 they are combined to retain their strengths and overcome their weaknesses.

Table 1 shows the distribution of the size of the blocks containing non-zero-initialized data for the SPECint2000 benchmark suite. Note that about one fifth of the statically allocated data contains code or data addresses, of which more than 85% is located in read-only data sections. Many of the data blocks contain at most one or two addresses. In blocks that are 16 bytes large, the last 8 bytes are very often padding and so contain no real data or addresses. It is clear that most of the blocks are small enough to significantly restrict the possible uses of data addresses.

4. GLOBALLY UNIFORM CONSTANT PROPAGATION

As shown in [8], aggressive global optimization techniques, such as constant propagation, achieve good results for code compaction. One reason is that at link-time, address calculations are optimization candidates as well. Indirect data accesses and indirect control flow transfers can often be transformed into direct data accesses and direct control flow transfers. This makes behavior of the program more explicit, thereby creating other optimization possibilities. As a side benefit, the addresses stored in memory for such indirect references often become dead and can be eliminated.

As discussed earlier, the detection of unreachable code and of dead data are closely related. We believe that this relation is of such a strong nature that a unified approach is required to obtain good results. The unified approach discussed in this paper consists of algorithms targeting simultaneously at

- the removal of unnecessary data accesses;
- the removal of dead data;
- a more accurate unreachable code elimination.

To achieve these goals to a large extent we use a unified approach based on constant propagation over the whole program. Constant propagation is a well-known data-flow analysis [2] carried out via a fixpoint computation over the control flow graph of a program. During the computation, variables at program points are mapped to lattice elements modeling the values they can hold. The lattice consists of all the possible constant values, $\top$ (undefined, meaning that we cannot say whether a variable holds a constant value or not, since we have not yet seen any definitions of the variable), and $\bot$ (non-constant, meaning that we have seen enough different definitions of a variable to decide that it may not have a constant value). Throughout this paper, we assume an optimistic constant propagator, where initially, the analysis assumes that all variables at all program points are undefined, i.e., $\top$, (except for the program entry point, where input values are unknown and mapped to $\bot$, i.e., non-constant). During the calculations, the mappings are lowered from $\top$ to constants or possibly to $\bot$. The algorithm finishes when the mappings have converged. When two or more different propagated mappings meet (this can happen where edges in the control flow graph have the same tail node), this results in $\bot$ being propagated from that point on.

It is well known that extending the basic analysis (a.k.a. *simple constant propagation*) to *conditional constant propagation* can be more effective in eliminating unreachable code than running unreachable code elimination as a separate phase [5, 24]. The difference between simple and conditional constant propagation is in the handling of conditional branches. During simple constant propagation, mappings are always propagated to both paths following the conditional branch. During conditional constant propagation by contrast, if a conditional branch's condition is mapped to a constant, allowing us to determine which path will be taken, the values are propagated over that path only. Conditional constant propagation, being a more effective combination of control and data flow analysis, is one example of the possible strength of a unified approach to optimization problems. The algorithms discussed in this paper build further on this, by extending conditional constant propagation.

One way to look at the extension to conditional constant propagation is to consider the worst-case assumptions (WCA) that are made before and during the analysis. These assumptions are made to guarantee soundness (i.e., correctness of the program transformations following the analysis) and termination of the analysis. For simple constant propagation, a single *a priori* worst-case assumption is made with respect to all conditional branches:

WCA 0 : Both paths following conditional branches are viable.

In the extended algorithm, making this assumption is deferred until later and split into separate, less conservative assumptions. More precisely, a separate assumption is now

is read in will be considered to be able to access any block where an address is taken, which will include the location whose address was passed to it.

made for each conditional branch, at the time appropriate to make the assumption for that particular branch:

WCA 0′ : Both paths following a conditional branch are viable when its condition becomes non-constant ($\perp$).

This approach of moving and refining worst-case assumptions is used in the rest of this paper to explain how our algorithms work. Starting from the worst-case assumptions necessary for a link-time conditional constant propagator, we will come to a constant propagator that to a large extent achieves the goals stated at the beginning of this section. A more formal description of the algorithms discussed in this paper can be found in [7].

4.1 Basic Link-Time Constant Propagation

Constant propagation during or after linking differs considerably from constant propagation at compile-time. There is no notion of variables, but instead registers and memory locations contain data. Statements in some high-level language are replaced by assembly instructions. All the dirty pointer arithmetic that a programmer uses implicitly is now explicit and has to be dealt with. As a consequence, alias analysis on binary programs is very difficult [9], resulting in a first *a priori* WCA to be made:

WCA 1 : All loads and stores potentially alias.

The consequence of this assumption is that we cannot propagate constants through memory. Constant propagation is limited to register contents only[2].

Three additional worst-case assumptions are made concerning the statically allocated data of a program:

WCA 2 : Statically (i.e. at compile or link-time) stored code addresses in live memory locations result in reachable code at those addresses.

WCA 3 : All memory locations containing statically allocated data are live.

WCA 4 : All statically allocated data in writable data sections are non-constant.

WCA 2 results from acknowledging that we do not have a sound and complete analysis of how these code addresses will be used by the program once we assume that they can be loaded. Therefore we conservatively assume that these code addresses can be used as targets for indirect control flow transfers. As WCA 3 states that all statically allocated data is live, and as this includes all the statically stored code addresses, this has dramatic effects: all statically stored code addresses result in program points becoming reachable and as a result not eliminable from the program. Moreover, as we do not know where the code addresses are loaded into the program and where they are used, the contexts in which the code at these addresses is executed is unknown. This means we have to initialize all registers with the value $\perp$ at these points.

WCA 3 above is (obviously) overly conservative; how to relax it is precisely the topic of this section.

WCA 4 results in the fact that no data from writable data sections will be propagated into the program by evaluating load instructions that load data from these sections. By contrast, load instructions loading from constant addresses (i.e. that have constant address operands) in the read-only sections are evaluated and the constant data these instruction load is propagated into the program by the constant propagator. This can later lead to the elimination of the load instruction and ultimately to the fact that the data becomes dead and can be eliminated from the program.

WCAs $0'$ and $1-4$ allow the implementation of a conservative but relatively good constant propagator. It is described and evaluated in more detail in [8].

4.2 Globally Uniform Constant Propagation

The constant propagator is now extended by moving and refining the worst-case assumptions. The refinement process is split in two steps: in a first step WCAs 3 and 4 are deferred and refined. In a second step, part of the refined WCAs is moved ahead in time again.

WCA $0'$, WCA 1 and WCA 2 will however remain unchanged as we have not yet found more restricted worst-case assumptions. WCA 1 is the reason for calling our final constant propagator the Globally Uniform Constant Propagator (GUCP): the statically allocated data in the program is either considered constant throughout the whole execution of the program or not-constant (and thus unknown) at all. This resembles the uniform division used in off-line partial evaluation theory [15].

Step 1. Refinement of WCAs 3 and 4

WCA 3 is replaced by three weaker assumptions:

WCA 3.1 : A memory location is live when there is an instruction loading data from that location.

WCA 3.2 : All memory locations are live if there is a load instruction that loads from an unknown location.

WCA 3.3 : A data address statically stored at some location that is live or contains non-constant data because of any assumption other than WCA 3.1 causes the entire block containing that address to become live and, if in a writable section, is assumed to contain non-constant data.

WCA 3.1 speaks for itself. Whenever there is a load instruction that is not evaluable during constant propagation, because it does not load from a known (i.e. constant) address, we make the straightforward WCA 3.2 that all memory locations are live.

[2] There is one exception. Callee-saved register stores and restores are treated in a special way. The involved register values are propagated from the store to the restore directly. A conservative but quite effective stack-behavior analysis was implemented for this [8].

WCA 3.3 is comparable to WCA 2. Whereas code addresses statically stored in live memory locations result in reachable code because the code addresses can be used to transfer control to, data addresses statically stored in live memory locations result in other data becoming live and non-constant, since the statically stored addresses can be loaded and consequently used for load/store operations. In between loading the address and using it it for loads/stores, there might be calculations on the address. These calculations can however only result in addresses from the same data block as the loaded address. Note that WCA 3.3 has to be made only for locations that are live because of assumptions other than WCA 3.1. The reason is that if a location is live solely because of WCA 3.1, the address statically stored at that location will be propagated into the program and its use will be analyzed, during which the necessary assumptions will be made. This is not the case when the location is live because of one of the other assumptions or when the data at that location is non-constant. In the latter case, even if we know exactly which instructions might load from that location, we will not evaluate them.

WCA 4 is replaced in a very similar way:

WCA 4.1 : A memory location contains non-constant data if there is an instruction writing to that location.

WCA 4.2 : All writable memory locations contain non-constant data if there is an instruction writing to a unknown (i.e. non-constant) location.

WCA 4.3 : If a constant address is being written to some (known or unknown) location by some instruction, the entrie block containing that address becomes live and, if in a writable section, is assumed to contain non-constant data.

WCA 4.4 : When a non-constant (unknown) value is written to some (known or unknown) location, all memory locations become live and, if in a writable section, is assumed to contain non-constant data.

The first three replacements are very similar to the replacement of WCA 3. The reasoning behind WCA 4.4 is straightforward: the non-constant value is conservatively assumed to potentially be any data address. For each of them, a separate WCA 4.3 should be made, which is expressed by WCA 4.4.

While the original WCAs were all *a priori* assumptions, this is no longer the case for their replacements; they are all made during the constant propagation. As a result, it might be that at some time during the fix-point calculations of the constant propagation, the statically stored data at a specific location is considered constant and propagated into the program, while at some later moment it might turn out that that data may not be constant. The instructions that loaded the original data into the program therefore have to be reevaluated, now loading $\perp$ into the program. To be able to keep track of these instructions efficiently, each writable memory location of which the data is assumed to be constant has associated with it a set of instructions. Every time a load instruction is evaluated and loads data from such a location into the program, the instruction is added to the corresponding set. When the location turns out to contain a non-constant, the instructions in its set will be reevaluated, now propagating $\perp$ into the program.

The major consequence of these replacements is that upon intialization of the constant propagation, the statically allocated data is considered dead. This means that the program entry point is considered the only reachable point upon initialization. It is only when memory locations containing code addresses become live that additional program points have to be assumed reachable (because of possible indirect control flow transfers) and reached by non-constants.

It is however clear no gain is to be expected from the altered algorithm so far, since WCAs 3.2, 4.2 and 4.4 will have to be made somewhere during the fix-point calculations for all non-trivial programs. The main reason is that when a non-constant is consumed by a load or store instruction, we have to assume that this could be any address, resulting in very pessimistic WCAs. Fortunately, they can be avoided by making WCAs when the non-constant values are produced rather than when they are consumed.

Step 2. From consumers to producers

At the point where a non-constant value is produced, i.e. when we loose track of a constant address during the propagation for some reason, the assumptions made about what that address will be used for can be limited to the address' data block. There are only a limited number of cases where constant addresses can get lost:

1. When a live (i.e. that will be consumed by some instruction) constant address meets a non-constant or another constant.
2. When an instruction derives a non-constant address from a constant address (e.g., when a non-constant index is added to a constant base address.) The resulting address will be propagated as a non-constant.
3. If at some indirect control flow transfer, the potential targets are not known and a constant address reaches the transfer.

If appropriate WCAs are made on all these occasions, there is no more need to make assumptions when non-constants reach load or store instructions. WCAs 3.2, 4.2 and 4.4 can therefore be replaced by one single assumption that has to be made if any of the three preceeding cases occurs:

WCA 5 : Whenever a constant address is no longer propagated as a constant, all locations in the address' block become live and, if in a writable section, they contain non-constant data.

Finally all WCAs are converted into much more restricted assumptions that are limited to single data blocks.

4.3 Discussion

As we put forward some goals for this algorithm, it is useful to evaluate its performance. It turns out that it performs

```
int a[3],b[3];

void f(int x,int y) {
  int* p;
  if (x) p=a;  else p=b;
  if (y) p+=1; else p+=2;
  return *p;
}
```

Figure 2: A simple code fragment in C-code

far from optimal. The main reason is the first case of the above enumeration.

Consider the code fragment in Figure 2: Depending on the argument `x` an array is selected. Depending on `y` an element from that array is selected. At two program points, where `a` and `b` are assigned to `p`, constant addresses are produced. These address are propagated and meet after the first if-then-else construct. The result is that $\perp$ is propagated from that point on. This properly captures one aspect of the computation—that the result is not a fixed constant address—but at a tremendous cost in precision, since the whole data block containing the arrays is now considered not only live, but also containing non-constant data.

In general, the problem is that when an address is lost because of a meeting, the absolutely worst-case assumption is made for its whole block with respect to its liveness and non-constant character. This in turn has a significant adverse effect on the precision of the overall analysis. In practice, almost all constant addresses propagated through the program somewhere meet other constants or non-constants. Assuming the worst-case scenario for such an address, that there will be loads from and writes to its whole block, is much too conservative: sometimes the address or derived addresses are only used by load instructions and not by stores (this also holds for unknown values calculated from constant addresses, as in the case of indexing a constant base address with an unknown index). Even more importantly, the address will often not be used to access all the locations in its block, but only some of them.

Basically, the constant propagator described here is comparable to monovariant partial evaluation. It is well known that polyvariant partial evaluation is more precise [15]. It is also much harder to implement because of efficiency and termination issues. In our case, fortunately, it is not necessary to partially evaluate the whole program, since we are only interested in what happens with the addresses. Furthermore, we know that calculations on addresses can only result in a fixed number of other addresses: they are always limited to the block the original address points to. This greatly simplifies a possible termination problem.

5. PARTIAL EVALUATION OF ADDRESS CALCULATIONS

The goal of partial evaluation of address calculations is, again, the detection of dead memory locations and constant data, avoiding the weak point of the constant propagator, i.e. the overly conservative assumption made when addresses meet other values. Avoiding the meeting in the example code fragment discussed at the end of the previous section can be done by propagating the two produced addresses in two completely separate propagations.

Consider the propagation of the address produced by the statement `p=a` in Figure 2. A GUCP initialized with this program point as the virtual program entry point will not propagate the address of `b`, so the address of `a` will not meet with that of `b`. After the second if-then-else the address of `a[1]` will however meet the address of `a[2]`, producing exactly the same result as the GUCP. A similar reasoning holds for a separate propagation of the address of `b`.

The difference is however that, if we are able to exclude the propagation of addresses other than those derived from `a` during its propagation, we know that the $\perp$ resulting from the meeting can only point to the block containing `a` instead of to the whole data memory. Propagating this produced $\perp$ further and postponing making any assumption until the $\perp$ is consumed circumvents the need for conservatively assuming at once that a location is live and holds non-constant data.

Since this separate propagation is still monovariant $\perp$ can be produced as a result of addresses that meet, as in the example. Such a $\perp$ will result in a WCA when it is consumed. While other addresses are not propagated during a specific separate partial evaluation (to avoid meetings), other constants are still being propagated (e.g. indices). When these constants meet, the result is a $\perp$ as well, but in his case the $\perp$ cannot be an address. So when it is consumed by a load or store instruction, there is no need to make any WCAs. To be able to differentiate between $\perp$ possibly being an address or not, a new tag for each register is used and propagated along with the lattice elements. This tag indicates whether or not the propagated lattice element is derived from the starting address the partial evaluation is performed for.

The partial evaluation of address calculations works as follows:

1. Mark all instructions directly reachable from the program entry point.
2. For each marked instruction producing a constant address according to the GUCP, apply a partial evaluation with that instruction as the only starting point.
3. During each single partial evaluation, a number of assumptions may have to be made, as was the case with the GUCP. How these assumptions differ is discussed below.
4. If during a single partial evaluation, instructions are evaluated that produce constant addresses (according to the GUCP), do not propagate these addreses, but propagate $\top$ instead. This avoids a large number of meetings, thereby avoiding the corresponding overly conservative assumptions. For these produced constant addresses, a separate partial evalution will be performed anyway.
5. During a single partial evaluation, load instructions might be encountered that now load from a known address, while the GUCP did not find them to do so. This can be the case when fewer paths leading to a load instruction are taken into account, just like meetings are avoided by not taking some paths into account. In

these cases, the load instructions are only evaluated when we have the *a priori* knowledge that the location they load from contains constant data. The reason for this is that for time and memory efficiency concerns, we only allow one partial evaluation per constant address producing instruction. This prohibits assuming data is constant until shown otherwise.

6. As opposed to constant propagation where propagation of constants basically ends when something becomes a non-constant, the partial evaluation continues with the propagation of register contents as long as derived (possibly non-constant) addresses are being propagated and new uses can be detected. This is the main reason we prefer to call this a partial evaluation: tags are propagated along with the values and termination depends on the tags as well as on the values being propagated. Moreover, only selected values will be propagated through a limited part of the program.
7. During one partial evaluation, a number of data locations become live. Code addresses statically stored at those locations result in additional instructions becoming reachable. These are added to the original set of reachable instructions and a separate partial evaluation is performed for all those additional instructions producing constant addresses, resulting in an iterative process.

How are the WCAs affected? It is clear that assumptions being made during a single partial evaluation can only effect the block containing the address produced by the starting point of the partial evaluation, since the only addresses propagated are addresses derived from that specific address. This means that there is no need to replace WCAs 3.2, 4.2 and 4.4 by WCA 5. Instead they are replaced by

A $3.2'$: All memory locations in the starting address' block become live if there is a load instruction loading from a non-constant address derived from the starting address.

A $4.2'$: All locations in the starting address' block contain non-constant data if there is a store instruction writing to a non-constant address derived from the starting address.

A $4.4'$: All locations in the starting address' block become live and contain non-constant data if an (constant or not) address derived from the starting address is written to some (known or unknown) location.

The occasions where a non-constant derived address reaches a load or store instruction are taken care of by the above assumptions. Constant derived addresses reaching loads or stores are still handled by WCA 3.1 and WCA 4.1. This is all done when constant or derived non-constant addresses are consumed, thereby avoiding the overly conservative assumptions for meeting in the GUCP.

What is left are the occasions when derived addresses (constant or not) are no longer propagated for some reason. Consider the cases where addresses were no longer propagated during a GUCP. These are enumerated at the end of subsection 4.2. The first two items of that enumeration are taken care of by using the tags: they are still propagated during a partial evaluation, albeit as tagged non-constants. WCAs $3.2'$, $4.2'$ or $4.4'$ have to be made when those reach load or write instructions.

The last case is not yet taken care of, so we still need to make a WCA in this case:

WCA $5'$: Whenever a derived address (constant or not) is lost during propagation because we don't know all the potential targets of an indirect control flow transfer, the whole starting address' block becomes live and, if in a writable section, it contains non-constant data.

Notice that the word "constant" in WCA 5 has been replaced by "derived" in WCA $5'$.

6. COMBINING THE TWO ANALYSES

Basically, both analyses result in a conservative approximation of the sets of data that are dead or constant. On the one hand, the result of the GUCP is hampered by the overly conservative assumptions made when addresses meet each other. On the other hand, the performance of the partial evaluation depends on the results of the GUCP: the more instructions the GUCP has found to produce constant addresses, the better the partial evaluation will perform, since more meetings will be avoided if the single propagation is being split into more separate partial evaluations.

However, each analysis is sound and stands on itself: that is, every live memory location is identified as such by each of the analyses; conversely, if either analysis identifies a location as being dead, then that location is definitely dead. The same reasoning holds with respect to some data being constant or not. To improve precision, therefore, we take the union of the two sets of dead locations: this results in a much larger set of dead data. Similarly, taking the union of the two sets of constant data results in a larger set of constant data. Since each analysis can benefit from the results of the other analysis, we iterate the two analyses until they converge.

7. EXPERIMENTAL RESULTS

For evaluating these algorithms, we have implemented them in Squeeze [8], a binary-rewriting tool that compacts binaries for the Alpha architecture. Squeeze achieves code compaction by two means: On the one hand it aggressively applies some well known interprocedural optimizations such as interprocedural constant propagation, context-sensitive liveness analyses, load-store avoidance, dead code elimination, unreachable code elimination, etc. On the other hand, Squeeze factors out code sequences that occur more than once in a program. Squeeze is based on Alto [18], a link-time optimizer oriented at speeding up programs.

The benchmark programs we used for evaluating our algorithms consist of the full SPECint2000 benchmark suite, two programs from the SPECfp2000 benchmark suite: 168.wupwise (Fortran 77) and 178.galgel (Fortran 90), five smaller C-programs from the MediaBench that are typical for embedded applications and 4 additional C++ programs using different libraries. Blackbox is a small, but fully functional window manager, Addressbook is a small GUI address book as found on PDAs (it is build using the Qt-library), GTL is

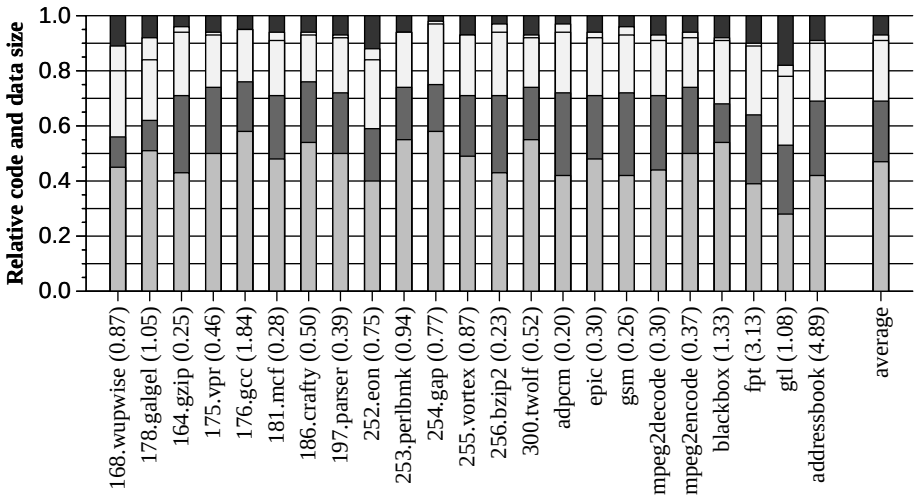

Figure 3: Program size reductions achieved
From bottom to top, the blocks in each bar represent the relative size of 1) code in the combined compacted binaries 2) data in the combined compacted binaries 3) code removed from the base binaries when applying code compaction only (4) additional code removed due to combined compaction and 5) data removed from the base binaries when using the combined compaction. Each program is annotated with its full size in MB.

the test program for the Graph Template Library, a C++ library with primitives for graph manipulation, and FPT is an in-house program that automates the process of parallelizing Fortran programs. FPT is a mixed language program written in C and C++.

The compilers we used to generate the binaries are Compaq's C V6.3-025, C++ V6.3-002 and Fortran X5.3 ECO2. These compilers use different vendor-supplied standard libraries, which is useful to show the generality of our techniques. All binaries were compiled with the -O1 flags (-O2 for Fortran) resulting in binaries that are optimized for code size. Our "base" binaries are produced by sending the compiler generated binaries through a base Squeeze version, eliminating no-ops and initial unreachable code (i.e. code that is not directly reachable and that does not have it address statically stored in the data). This means the base binaries are what might be obtained from a reasonably smart linker. The same profile-guided code layout and scheduling by Squeeze has been applied on all binaries compared in this section, to allow a fair comparison that is not hampered by differences in code scheduling and layout. The "combined" compacted binaries are compacted using all of Squeeze's compaction algorithms. The "code" compacted binaries are compacted using basically the same Squeeze version, but without performing the partial evaluation of address calculations and with the basic constant propagator as discussed in section 4.1 instead of the GUCP. As a result, no data was removed in the code compacted binaries.

For linking, we used the vendor-supplied linker with flags to produce statically linked executables containing symbol and relocation information, and to dump a map indicating where the blocks of the object files are located in the final binary. It is this map we use to divide the data section into blocks.

The overall code and program size reductions using our combined analyses are given in Figure 3. While the code compacted binaries were on average 22.9% smaller, program size reduction is 30.6% when the combined approach is used. This difference (7.7% on average, ranging from 1.8 to 20.0% for the individual benchmarks) results largely from the removal of dead data and less from additional elimination of code, as the additional gain in code size reduction is much smaller. On average 24.3% of the data is removed from the program, whereas the code size reduction is on average 33.6%. Without the combined analysis, this would have been 30.4%. Remember that these numbers do not take into account the initial unreachable code removal performed on all the binaries by the base version of Squeeze.

The results for some of the C++ programs, 252.eon and GTL, are quite remarkable. The compacted binaries are less than half the size of the original ones. The result is that the statically linked, compacted binaries are 17% (252.eon) and 33% (GTL) smaller than the dynamically linked ones! The reason is that the dynamically linked program consist for a large part of a dynamic string and symbol table.

Table 2 compares the execution times for the SPECint2000 base programs, the base programs with profile-directed code layout added, and the programs resulting from Squeeze. The experiments were run on a Compaq DS20E AlphaS-

tation with two 667 MHz 21264 EV67 processors, 8 MB L2 DDR cache per processor and 1.5 GB of RAM, running Tru64 Unix 5.1. It can be seen that the compaction of code (and data) typically does not come at the cost of speed: e.g., for the SPECint-2000 benchmarks both the code and combined compacted programs are, on the average, about 8% faster than the base programs.

The cost of applying the combined analysis is relatively low: SQUEEZE on average requires about 10% more time to compact the binaries when the proposed algorithms are invoked, while the additional memory requirements are modest. Information relating to the dead and read-only character of a location can be stored in 2 bits. The sets associated with memory locations that hold instructions are linear in the program size as well, as each instruction can only be in one set, since it can load from at most one constant address. The total amount of additional memory required for these analyses therefore is only a small fraction of the total memory footprint of the data structures required by other analyses in SQUEEZE.

8. RELATED WORK

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 [10, 11, 12, 13, 16, 17, 19]. 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 [10, 11, 12, 13]—which can be problematic for limited-memory devices—or require special hardware support for executing the compressed code directly [16, 17]. By contrast, programs compacted using our techniques can be executed directly without any decompression or special hardware support.

Most of the previous work on code compaction to yield smaller executables treats an executable program as a simple linear sequence of instructions [3, 6, 14]. They use suffix trees to identify repeated instructions in the program and abstract them out into functions. None of these works address the issue of reducing the size of the data section within a program. 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 several bytecode instructions. 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.

We have recently showed that an alternative approach, using the conventional control flow graph representation of a program and based by and large on aggressive interprocedural compiler optimizations aimed at eliminating code, can achieve significant reductions in code size, averaging around 30% [8]. However, this work does not take into account the removal of dead data, and the synergistic effect this has on the removal of unnecessary code. The work we have reported in this paper yields code size reductions that are on average 33.5%. The elimination of unused data from a program has been considered by Srivastava and Wall [21] and Sweeney and Tip [22]. Srivastava and Wall, describing a link-time optimization technique for improving the code

binary	base	code	combined	$\frac{code}{base}$	$\frac{combined}{base}$
164.gzip	513	476	483	0.93	0.94
175.vpr	431	414	413	0.96	0.96
176.gcc	270	259	261	0.96	0.97
181.mcf	481	471	468	0.98	0.97
186.crafty	253	218	226	0.86	0.89
197.parser	787	746	730	0.95	0.93
252.eon	379	307	310	0.81	0.82
253.perlbmk	349	370	381	1.06	1.09
254.gap	421	408	410	0.97	0.97
255.vortex	547	414	407	0.76	0.74
256.bzip2	439	397	390	0.90	0.89
300.twolf	687	638	628	0.93	0.91
geometric mean				0.92	0.92

Table 2: Execution times (in seconds) and ratios for the base, code compacted and combined compacted binaries of the SPECint2000 benchmark suite.

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. The work of Sweeney and Tip is restricted to eliminating dead data members in C++ programs, and so is not applicable to non-object-oriented programs; by contrast, our approach, which works on executable programs, can be applied to programs written in any language. Neither of these works addresses the close relationship between the elimination of data and the elimination of code. Sweeney reports a size reduction of 4.4% on the average; by considering the elimination of both code and data, by contrast, we achieve size reductions of 23.6–46.6% overall.

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] such systems obtain higher compaction levels than our system. They are however programming-language specific and start from programs containing the whole run-time environment of Self applications. For Java [23] similar results to ours are achieved, but again the techniques used are language specific.

9. CONCLUSIONS AND FUTURE WORK

Because of the growing deployment of mobile and embedded processors with a limited amount of available memory, techniques that reduce the memory footprint of programs are becoming increasingly important. Previous work on this topic has typically focused either on the reduction of data areas or on reduction of code areas, but not on both, even though there are obvious dependences and synergies between the two. This paper describes a low-level analysis that reasons about the use of code and data addresses within programs, and thereby is able to exploit these dependences and synergies. The proposed algorithms are generally applicable and not limited to a specific programming language or a particular implementation context. Experimental results indicate that the resulting system achieves significantly better memory footprint reductions than previous work.

The algorithms proposed in this paper can be refined in a number of ways. One place where the combined algorithm

loses precision is due to the worst-case behavior when addresses are stored. This can be improved—at least for stack saves and restores—by detecting where addresses saved on the stack can be loaded into the program again. Moreover, using a polyvariant partial evaluation for each produced address will produce more precise results as well.

The algorithms can be applied using whatever kind of data blocks that conform to the rule that an address pointing to one block cannot be derived from an address pointing to another block. Interval-analysis could be used to split the blocks we use today in smaller blocks. Compilers could assist this process as well, e.g. by indicating borders in the data sections of object files that are not crossed by address computations. They might even produce multiple object files for each source code file. All statically declared objects that have no overlap with other objects in memory can be put in another object file. This might occasionally result in less efficient object code because the compiler does not know the relation between the addresses of those objects. Link-time optimizers such as ALTO or SQUEEZE will easily remove these inefficiencies though. Preliminary research has shown us that the very similar approach of automatically splitting source code files into multiple files that define only one object, results in no additional compaction. While our link-time code compaction system was able to remove most inefficiencies introduced by the compiler due to its more limited view on the program during compilation of one such splitted file, it was almost never capable of producing significantly smaller binaries.

10. REFERENCES

[1] O. Agesen and D. Ungar. Sifting out the gold: Delivering compact applications from an exploratory object-oriented environment. In *Proc. OOPSLA*, pages 355–370, Oct. 1994.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers, Principles, Techniques and Tools*. Addison-Wesley, 1986.

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

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

[5] C. Click and K. Cooper. Combining analyses, combining optimizations. *ACM TOPLAS*, 17(2):181–196, March 1995.

[6] K. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. PLDI*, pages 139–149, May 1999.

[7] B. De Sutter, B. De Bus, S. Debray, and K. De Bosschere. Combining global code and data compaction. Technical Report PARIS 01-02, Ghent University, 2001.

[8] S. Debray, W. Evans, R. Muth, and B. De Sutter. Compiler techniques for code compaction. *ACM TOPLAS*, 22(2):378–415, March 2000.

[9] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Proc. 1998 ACM Symposium on Principles of Programming Languages (POPL-98)*, pages 12–24, January 1998.

[10] J. Ernst, W. Evans, C. Fraser, S. Lucco, and T. Proebsting. Code compression. In *Proc. PLDI*, pages 358–365, June 1997.

[11] 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.

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

[13] C. Fraser. Automatic inference of models for statistical code compression. In *Proc. PLDI*, pages 242–246, May 1999.

[14] C. Fraser, E. Myers, and A. Wendt. Analyzing and compressing assembly code. In *Proc. ACM SIGPLAN Symposium on Compiler Construction*, volume 19, pages 117–121, June 1984.

[15] N. Jones, C. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall International, 1993.

[16] 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), November 1998.

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

[18] 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, January 2001.

[19] W. Pugh. Compressing Java class files. In *Proc. PLDI*, pages 247–258, May 1999.

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

[21] A. Srivastava and W. Wall. Link-time optimization of address calculation on a 64-bit architecture. In *Proc. PLDI*, pages 49–60, June 1994.

[22] P. Sweeney. and F. Tip. A study of dead data members in C++ applications. In *Proc. PLDI*, pages 324–323, June 1998.

[23] F. Tip, C. Laffra, and P. Sweeney. Practical experience with an application extractor for Java. In *Proc. OOPSLA*, pages 292–305, Nov. 1999.

[24] M. Wegman and F. Zadeck. Constant propagation with conditional branches. *ACM TOPLAS*, 13(2):181–210, April 1991.