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

More and more computers are being incorporated in devices where the available amount of memory is limited. As a result research is increasingly focusing on the automated reduction of program size. Most if not all of the literature in this area either focuses on code compaction or on the removal of dead data. They are however closely related as code addresses are nothing but data. The main contribution of this paper is to show how combined code and data compaction can be achieved using a link-time code compaction system that works by reasoning about the use of both code and data addresses. The analyses proposed are built on fundamental properties of linked code and therefor generally applicable. The combined code and data compaction is evaluated on SPEC2000 and MediaBench programs, resulting in binary program size reductions of 24.0%–45.8%. This compaction involves no speed trade-off, as the compacted programs are on average about 5% 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, both because of the additional functionality that they provide, and because of 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 retaining the property that they can be executed directly without requiring any decompression. Most of the literature on compaction focuses on either code or data compaction. Data compaction research is limited to simple literal address removal from object files [15] or the removal of dead data members in OO-languages, where the analysis is applied on source code, thus forcing conservative assumptions about library code [16]. Work on code compaction has generally focused on identifying repeated instruction sequences within a program and abstracting them into functions [3, 9].

In all of this work, data and code compaction have been carried out independently of each other. It is not difficult to see, however, 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 [14, 13]). Code optimizations such as dead and unreachable code elimination can cause data to become unreachable as well, 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 potentially eliminable, as well. 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. 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 uniformly and 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 resulting system achieves size reductions that are significantly better than have been reported in the past: for example, on the SPECint-2000 benchmark suite, we achieve reductions of about 35%–40%, on the average, in the number of program instructions, and 27%–32% in the total program size (instructions+data). Our ideas rely only on general properties of compiled code and so is not restricted to a particular implementation context. For simplicity of exposition the discussion below will focus on load-store architectures, where arithmetic operations involve only registers, and memory is accessed only via `load` and `store` instructions; however, the ideas presented here are not lim-

*The work of B. De Sutter, K. De Bosschere and B. De Bus was supported by the Fund for Scientific Research – Flanders under grant 3G001998. The work of S. Debray was supported in part by the National Science Foundation under grants CCR-0073394, EIA-0080123, and ASC-9720738.

ited to such architectures, and can be readily adapted to architectures supporting more complex addressing modes.

2. STRUCTURE OF COMPILED CODE

The object module generated by a compiler from a source module typically consists of several code and data sections; examples of such sections include the code section, the constant data section, the zero-initialized 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 of a base address). This register is then used as a source operand to access that location. Now consider the locations that an address computed in this way could possibly refer to. 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. It therefore 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 stores, the data is read-only.

This property is fundamental to the analyses described later in this paper, in Sections 3 and 4. Both analyses are able to detect dead and read-only memory areas, and each algorithm has its strengths and weaknesses. In Section 5, they are combined to retain their strengths and overcome their weaknesses.

Table 1 gives some insight in 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. Note how 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 put severe restrictions on the possible uses of the data addresses.

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.

3. GLOBALLY UNIFORM CONSTANT PROPAGATION

As shown in [4], aggressive global optimization techniques, such as constant propagation, achieve good results for code compaction. One of the reasons for this success is that at link-time address calculations are candidates for optimization as well. Indirect data accesses and indirect control flow transfers can often be transformed into direct data accesses and direct control flow transfers. The behavior of the program then becomes more explicit, thereby creating other code optimization and compaction possibilities. As a side benefit, the addresses stored in memory for the indirect data accesses and control flow transfers often become dead because they are no longer loaded after this transformation.

As constant propagation (of addresses) is the driving force behind these transformations and code compaction, we extend constant propagation to achieve the following goals:

- Detection of read-only data, which helps us refine the control flow graph of the program by allowing us to resolve the possible targets of indirect control transfers.
- Detection of dead data that can be removed from the program. A potential side benefit is that the removal of such data can result in fewer possible indirect control transfer targets and less indirect accessible data.
- Resolution of the possible targets of indirect control transfers on the fly, i.e., during the analysis itself, since this generally yields better results than doing it in a separate phase. This is comparable to conditional constant propagation, which basically performs on-the-fly unreachable code elimination and performs better than separate simple constant propagation and unreachable code elimination [2].

3.1 Basic Constant Propagation

Figure 1 shows the pseudo-code for a basic constant propagation algorithm (the reader interested in a deeper treatment of constant propagation is referred to standard texts on optimizing compilers, e.g., [17]). Here i denotes an instruction, r a register and m a memory location or address.

[1] It is possible, in principle, for a program to communicate such pointers from one point in a program to another in non-standard 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 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.

```
BasicConstantPropagation():
  Init()
  Fixpoint()

Init():
  for all i, r : InsMap[i, r] = ⊤
  for all r : InsMap[program entry point, r] = ⊥
  MarkIns[program entry point] = TRUE

Fixpoint():
  while(∃i : MarkIns[i] == TRUE)
    for all i with MarkIns[i] == TRUE :
      MarkIns[i] = FALSE
      Propagate(i, Evaluate(i))

Meet(x, y):
  return x ⊓ y

Evaluate(i):
  switch (type(i))
    case Op :
      return SymbExe(i)
    case Load :
      let the address being loaded from be m;
      if (m is a constant address ∧
             Block[m] in constant section)
        return SymbExe(i)
      else
        return InsMap[i] with destination register set to ⊥
    case Store :
      return InsMap[i]

Propagate(i, tmp):
  for all successors j of i :
    if (Meet(tmp, InsMap[j]) ≠ InsMap[j])
      InsMap[j] = Meet(tmp, InsMap[j])
      MarkIns[j] = TRUE
```

Figure 1: A simplified basic constant propagation algorithm.

`InsMap`$[i, r]$ is the lattice element mapped to register r at the program point of instruction i. `InsMap`$[i]$ is the array of all register value mappings at that point. `MarkIns`$[i]$ is a boolean indicating whether the fix-point algorithm should re-evaluate the instruction. `SymbExe`(i) returns the register content mappings after symbolic execution of the instruction on its mapping. `Block`$[m]$ is the data block containing the memory location m.

The lattice this fix-point algorithm uses is depicted in Figure 2. `InsMap`$[i, r]$ is mapped to $\top$ if register r has not been defined at program point i; it is mapped to C_i if the register holds that constant; and to $\bot$ when it (possibly) does not hold a constant value. The constant propagation algorithms we use are optimistic: before the fix-point calculations all register contents at all program locations are assumed to possibly be constants, except for the values at the program entry point.

The basic constant propagation shown in Figure 1 is kept as simple as possible for the sake of clarity. Our implementation uses an aggressive context-sensitive interprocedural constant propagator. It works on a low-level intermediate representation of executable programs and so it is limited to the propagation of register contents. No data is propagated through memory locations, except for data in constant data sections: if they are loaded by instructions with constant source operands, the data is propagated into the program. When possible, a conditional branch based on the value of a register r propagates information about r into the successor blocks: for example, an instruction '`beq r, ...`,' which branches if register r is 0, propagates the information that r has the value 0 into its *true*-branch. If the register tested by a conditional branch evaluates to a constant value (i.e., the corresponding test has a fixed known outcome), the control flow edge that is not taken is discarded.

3.2 Globally Uniform Constant Propagation

Given our assumption that values stored in memory can only be accessed via `load` instructions, it can be seen, from the pseudo-code for `load` instructions (in `Evaluate()`) in Figure 1, that only data values from constant data sections will be propagated into the program. It is also clear that this propagation does not give information about the liveness or read-only character of the data in writable data sections. To address these shortcomings, we extend the basic constant propagator in four ways:

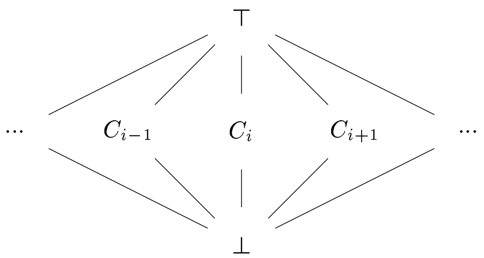
1. all statically allocated global data is assumed constant and dead at the start of the fix-point algorithm,
2. statically allocated global data that is accessed somewhere during constant propagation is marked as live,
3. statically allocated global data that is written somewhere during constant propagation is marked as writable,
4. conditional constant propagation is extended to indirect control flow transfers.

To formalize this, we need a lattice for the memory locations. This lattice has the same structure as that shown in Figure 2, but the lattice elements now have different meanings. If a memory location is mapped to $\top$, this means that the location is dead, i.e., it cannot be used by the program and we don't care what it contains. Mapping a location m to a value C_i denotes that m contains the value C_i and m may be read by the program, i.e., m is live. If a location is mapped to $\bot$, this denotes that the program may write to this location, so for the rest of the propagation we don't know what value is stored there.

This lattice explains why the extended propagator is called the Globally Uniform Constant Propagator. Statically allocated global memory locations are considered to have a constant value throughout the execution of the whole program or are considered non-constant. This resembles the uniform division used in simple off-line partial evaluators [10]. Note that the same lattice is used for two different things: for mapping register contents and for mapping memory locations.

The extended algorithm is given in pseudo-code in Figure 3. New or changed lines are indicated with a '$-$' in the left margin. In this code `MemMap`$[m]$ is the lattice element mapped to memory location m. `MemRefSet`$[m]$ is the set of instructions that during constant propagation loaded the data at location m.

Initially, all data locations are considered to be dead by the algorithm (mapped to $\top$ in line 4 of `Init`), and no loads of data are considered to have occured (line 5 of `Init`). The algorithm then iteratively identifies locations that may be live. When a load instruction is evaluated, if the source



$$\bot \sqcap C_i = \bot$$
$$\top \sqcap C_i = C_i$$
$$C_i \sqcap C_j = \bot \text{ iff } i \neq j$$
$$C_i \sqcap C_j = C_i \text{ iff } i = j$$

Figure 2: The lattice used for CP and the meet rules

operand is a constant address and the corresponding memory location is not mapped to $\bot$, the statically allocated value at that address is loaded and propagated into the program. Because it may later turn out that this value cannot be guaranteed to be a constant, we add the instruction to the set of instructions that loaded from this memory location (`MemRefSet[`m`]`). This happens in the `Load` case in `Evaluate`. This is fundamentally different from the basic constant propagator, since the edges of the control flow graph are no longer the only links that control which instructions should be re-evaluated after lowering a register mapping at some program point.

If we discover a store to some constant location in `Evaluate`, `WriteMem` sets the mapping of that location to $\bot$. This means that we assume worst-case behavior for this memory location: there can be loads and stores from and to it. There might be loads in the program from that address that we will not evaluate. Therefore, if the statically allocated value at that location is a data address itself, we have to assume that this address may be used for loads and stores as well. In the algorithm, the recursive call to `WriteMem` takes care of this. Note that a statically allocated value in memory is an address if and only if it is relocatable. Our implementation uses relocation information to distinguish between ordinary data and addresses.

At all times during the propagation, if a constant address is being propagated and at some point during the analysis we lose track of exactly which address we are working with (e.g., due to address arithmetic where one of the operands may not be known), we make the worst-case assumption that the program will write in the whole block containing that address: `WriteMem` is executed on the whole block. We have to make this assumption because we don't track the use of this address any longer, and so must make worst-case assumptions about the ways in which it could be used. The worst-case assumption is stated in:

- `Meet`(x, y): if a propagated address meets another value or $\bot$ and thereby is no longer propagated as a constant.
- `Propagate`(i): if the successors of an indirect control transfer cannot be resolved at some program point, this is modeled with a special successor node *Unknown* in the control flow graph. This node is assumed to have worst-case behavior: it reads from and writes to all registers and all memory locations whose addresses are propagated into it; in particular, all registers are mapped to the lattice element $\bot$ at *Unknown*. Computing the meet of the propagated values with $\bot$ at this node assures that `Meet` takes care of unknown successors.
- `Evaluate`(i): if a constant address is stored, we don't track the use of that stored value, since our propagator notes only that the memory location is writable (by mapping it to $\bot$), and as a result will not load the contents of that address later in the analysis.

Note that stores where no constant addresses are involved—e.g., a store to an address that is loaded from a memory location whose contents cannot be predicted statically—do not have to be treated specially. This is because the only reason the address being stored to is unknown is that we lost track of the possible addresses that could reach the store instruction during propagation. As discussed above, when we lose track of an address we make worst-case assumptions about what may happen, so such stores are conservatively handled by one of the three cases above.

The fundamental reason why this algorithm works is the organization of memory into blocks, as discussed in Section 2. For a block B to be live, a pointer to that block must be loaded somewhere in the program. If we detect such a pointer, two things can happen: either we find all its uses and have an accurate picture of its use, or we lose track of the pointer somewhere and assume the worst-case scenario: the whole block can be written. If we don't detect a pointer to the block in the code, there are two possible reasons: either B is dead or it can only be accessed through data in some block B' but we don't know the contents of B'. In the latter case B' must have been written to at some point in the program or somewhere we had to make worst-case assumptions about the use of data in B', and the recursive call in `WriteMem` at that point handles this.

At the end of the constant propagation, all constants (including addresses) that are found are propagated into the program and dead blocks are marked for removal. Further optimization of the program may find that it is more efficient to compute some loaded values instead of loading them into a register (e.g., see [15]). Some data can additionally become dead if this happens, and can be removed from the program.

The fourth extension, namely the generalization of conditional constant propagation to indirect control transfers, permits on-the-fly resolution of indirect control flow transfers where possible. Recall that for conditional constant propagation, information at a conditional branch node in the control flow graph is propagated over only one outgoing edge if the condition of the branch evaluates to a constant. This yields better results than simple constant propagation followed by a separate pass of unreachable code elimination. The same holds for indirect control flow. Suppose that prior to constant propagation we don't know the target of an indirect branch at some program point: as mentioned

```
GloballyUniformConstantPropagation():
  Init()
  Fixpoint()

Init():
  for all i, r : InsMap[i, r] = ⊤
  for all r : InsMap[program entry point, r] = ⊥
  MarkIns[program entry point] = TRUE
– for all m : MemMap[m] = ⊤
– for all m : MemRefSet[m] = φ

Fixpoint():
  while(∃i : MarkIns[i] == TRUE)
    for all i with MarkIns[i] == TRUE :
      MarkIns[i] = FALSE
      Propagate(i, Evaluate(i))

Evaluate(i):
  switch (type(i))
    case Op :
      return SymbExe(i)
    case Load :
      let the address being loaded from be m
–       if (constant address m ∧
–            (MemMap[m] ≠ ⊥ ∨ Block[m] in constant section))
–         MemRefSet[m] = MemRefSet[m] ∪ {i}
–         MemMap[m] = Meet(MemMap[m], loaded value)
        return SymbExe(i)
      else
        return InsMap[i] with destination register set to ⊥
    case Store :
–     if (constant address m is stored)
–       for all n in Block[m] :
–         WriteMem(n)
–     if (x is stored at constant destination m)
–       WriteMem(m)
      return InsMap[i]

Propagate(i, tmp):
  for all successors j of i :
    if (Meet(tmp, InsMap[j]) ≠ InsMap[j])
      InsMap[j] = Meet(tmp, InsMap[j])
      MarkIns[j] = TRUE

Meet(m, n):
– if (m is a constant address ∧ (m ⊓ n ≠ m))
–   for all o in Block[m] :
–     WriteMem(o)
– if (n is a constant address ∧ (m ⊓ n ≠ n))
–   for all o in Block[n] :
–     WriteMem(o)
  return x ⊓ y

WriteMem(m):
– if (MemMap[m] is a constant address)
–   WriteMem(MemMap[m])
– MemMap[m] = Meet(MemMap[m], ⊥)
– for all i in MemRefSet[m] :
–   MarkIns[i] = TRUE
– MemRefSet[m] = φ
```

Figure 3: The Globally Uniform Constant Propagator algorithm.

earlier, this is modeled with the special successor node *Unknown* that enforces worst-case behavior. Now suppose that at some point during the fix-point iteration in the analysis, a target address A (i.e. the address of a possible successor) is loaded from a location A' that is still considered read-only. It seems likely that if the target address A reaches the indirect branch instruction, then so will the address A' from which it was loaded. However, if we simply propagate the register values to *Unknown*, the assumptions regarding its worst-case behavior—specifically, that it may write to all addresses propagated into it—will cause A' to be mapped to $\perp$. The result would be that the instruction loading the target address has to be re-evaluated, now with the block containing A' marked writable. The effect of this is that we can no longer infer that the target address loaded is A, and so lose the ability to resolve the target of the indirect branch.

The solution is to optimistically propagate the register lattice mappings to the successor at address A. If it turns out, during the rest of the fix-point computation, that the contents of block B (where the target address A was loaded from) cannot be overwritten on that path, then we have succeeded in resolving the target A of the indirect branch. If the target address is overwritten at some point in the computation, the instruction loading the target address will be re-evaluated with the block containing A' marked as writable, and we will correctly infer that A is not the only possible successor of the indirect branch.

It should be emphasized that the analysis presented above models only the code (InsMap) and the statically allocated data (MemMap) in the program, *not* the entire space of addressable memory. For this reason, the memory requirements of the analysis are quite reasonable. The space required for MemMap$[m]$ for a location m consists of a word for the C_i values and an additional byte for the possibilities $\top$ and $\perp$. In our current implementation, on a 64-bit architecture, this incurs only 12% more space than the amount of statically allocated data in the program.

3.3 Discussion

As we put forward some goals for this algorithm, it is useful to evaluate its performance. It turns out that the performance of the algorithm is quite poor. The problem is the Meet operation. Suppose that m and n are constant addresses and $m \neq n$, then Meet(m, n) will be computed as $\perp$: 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 lattice element $\perp$ for memory addresses is interpreted as a complete lack of information: that is, the blocks containing m and n are considered to possibly be read from or written to during execution. The problem with this is that it loses information about memory blocks that are read-only, which 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 in Meet. Assuming the worst-case scenario for such addresses, that there will be loads from and stores to their whole block, is much too conservative: it is often the case that there are only loads using many of these addresses.

Basically, the constant propagator described here is comparable to monovariant partial evaluation. It is well known that polyvariant partial evaluation performs much better. It is also much harder to implement because of efficiency and termination issues. In our case, fortunately, it is not necessary to fully partially evaluate a 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 solves a possible termination problem.

4. PARTIAL EVALUATION OF ADDRESS CALCULATIONS

The goal of partial evaluation of address calculations is, again, the detection of dead and read-only memory locations, avoiding the weak point of the constant propagator, i.e. the overly conservative `Meet`. As described below, each constant address that is produced is propagated separately by our partial evaluator: this makes it difficult to incorporate the resolution of indirect control transfers into this analysis. Because of this, we do not attempt to resolve indirect control transfers here, but instead rely on the results obtained from the constant propagation described in the previous section (the precision problem with `Meet` in the constant propagator notwithstanding).

Our partial evaluator works in three phases, as discussed below. The same memory lattice is used as in the constant propagation for memory locations and all memory locations are again initialized to $\top$.

Phase 1. Detection of Loads/Stores at Constant Addresses.
During the first phase, the program is scanned for instructions that load or store from or to constant addresses. These are the instructions for which the constant propagator has found constant address arguments. The memory lattice mappings are adjusted accordingly: if there is a load from a constant address A, then if the constant propagator indicates that location A contains a constant C then A is mapped to C, otherwise it is mapped to $\perp$; if there is a store to address A, then A is mapped to $\perp$.

Phase 2. Detection of Uses of Non-Constant Addresses.
In the second phase, the program is scanned for instructions that produce constant addresses. This is a subset of the instructions that are found by the constant propagator to have constant operands. In particular, we want to identify computations where a constant address A_0 is used to compute other addresses $A_1, \ldots, A_n$. For each of the addresses so computed, we want to keep track of the fact that they were derived from A_0.

To do this, we carry out a mono-variant binding-time analysis for each instruction I that produces a constant result, starting at I with its result as a static value. The specific notions of static and dynamic variables in partial evaluation theory will in the remainder of this section be called 'constant' and 'non-constant,' for consistency with the constant propagation algorithm discussed in the previous section. The lattice used here is the same as that for constant propagation of register values, i.e., $\top$ means that the register's value if undefined, a value C_i means that the register is guaranteed to contain the value C_i, and $\perp$ means that the value of the register may not be a fixed constant. The only difference is that register values at program points are initialized to constants (if the constant propagator has found them to be constant) or $\perp$ if they are not constant according to the constant propagator.

To identify addresses that are derived from other addresses, each register is also tagged with one of the elements `D` or `ND`, denoting, respectively, *derived* or *not-derived* from the initial address from which the binding-time analysis was initiated. They form a lattice with only two simple meeting rules:

$$\texttt{ND} \sqcap x = x$$
$$\texttt{D} \sqcap x = \texttt{D}$$

Partial evaluation proceeds as described below. Recall that at the beginning of partial evaluation, all addresses are mapped to $\top$, i.e., marked as dead. Some locations then have their mappings changed to a non-$\top$ value in Phase 1. Phase 2 then updates the mappings of yet more locations. When changing the mapping of an address during this process, we always set it to the meet of the old and new mappings for that address. Thus, if the old mapping of a location is x and we want to update it to a value y, the mapping of that location is set to $x \sqcap y$. Since $\sqcap$ is associative and commutative, this means that the order in which the updates are carried out does not affect the final result. To reduce repetition and simplify the presentation, the discussion below does not explicitly refer to this aspect of updating the lattice mappings.

- The same symbolic execution of evaluable instructions as in constant propagation is used.
- The value (constant or $\perp$) produced by an instruction is tagged with `D` or `ND` depending on the type of instruction and the tags of the instruction operands. For example, an `Add` instruction adding some value to a (constant or non-constant) value tagged `D` will result in a tag `D`, as this means that some value is added to an address derived from the original address, which results again in an address derived from the original address. A `Compare` instruction comparing a `D`-tagged value to something else produces an `ND` mapping, since the result of a comparison is not an address.
- In addition to the previous rule, the result of an instruction for which the constant propagator has found the produced value to be a constant, is tagged `ND`. If the produced value is an address, it will be propagated in a separate binding-time analysis. This is precisely how we avoid the problems of the `Meet` procedure during constant propagation.
- If at some program-point during partial evaluation, no registers are mapped to `D`, evaluation along that path stops, since there can be no more uses of the address or its derivatives along that path.
- If at some program-point, a register mapped to a constant address C_i is used as an address for a load or store, the lattice mapping of the memory location at that constant address is updated accordingly. In this case, however, even if the value that is loaded can be determined to be a constant address, it is not considered to be a derivative of the original address and therefore loaded as a non-constant.
- If at some program-point a constant address A is itself stored in memory, the whole block containing the address A has its mapping changed to $\perp$. In effect, we assume that since A is being stored into memory, the program may subsequently load the contents of this memory location and use it in ways that we cannot anticipate, so we make worst-case assumptions.

- Suppose that, during partial evaluation starting with a constant address A, at some program point we encounter a load from a non-constant address tagged with `D`. This means that there is a load from some address (whose exact value we don't know) derived from A. Based on our earlier assumptions (see Section 2), an address derived from A must refer to a location in the same block as A, we conclude from this that every location in the block containing A is live. The partial evaluator therefore maps each such address to the mapping for that address computed by the constant propagator (i.e., either a constant C_i or $\perp$).
- If at some program-point, a non-constant tagged with `D` is used in a store instruction, the whole block containing the original address is mapped to $\perp$. As this is the worst case, partial evaluation is finished.

The reason why this algorithm performs better on some places than our constant propagator is because the mono-variant partial evaluation is performed separately for each instruction producing a constant address. By performing multiple mono-variant partial evaluations, we approximate the result of a poly-variant partial evaluation and we avoid most of the meeting between constant addresses and other values or non-constants in the `Meet` procedure of the Globally Uniform Constant Propagator.

Phase 3. **Fix-point Detection of Accessible Data.** The final phase consists of a fix-point computation for the detection of accessible data. If a memory locations A is live and it holds an address A', then A' is assumed to be accessible as well. This is repeated until no new locations are found to be accessible.

5. COMBINING THE TWO ANALYSES

Basically, both analysis result in a conservative approximation of the sets of data that are accessible or read-only. The result of the Globally Uniform Constant Propagation was hampered by the overly conservative Meet procedure, while the partial evaluation suffered from indirect control flow transfers that it had to treat very conservatively. However, each analysis is sound: that is, every memory location that can be accessed is identified as accessible by each of the analyses; conversely, if either analysis identifies a location as being dead, then that location is definitely dead. To improve precision, therefore, we take the intersection of the two sets of accessible data: this results in a much smaller set of data that is inferred to be accessible. Analogously, taking the union of the two sets of dead data blocks results in a larger set of blocks being inferred as dead.

The two analyses are combined as follows:

- Each update of the memory lattice mappings during the constant propagation has as a lower bound the mapping found by the partial evaluation. Thus, if the partial evaluation maps a memory location A to a value x and the constant propagator wishes to update the mapping of A to y, then A is mapped to the value $x \sqcup y$.
- The constant propagation and partial evaluation are executed several times, on an interleaved basis: first the constant propagation, then the partial evaluation.
- Before the first constant propagation phase, the mappings that have not yet been computed by the partial evaluation are set to the worst-case values, i.e., all locations are writable.

language	compiler
C	Compaq C V6.1-011
C	gcc version 2.95.2 19991024
C++	Compaq C++ V6.2-024
Fortran 77	DIGITAL Fortran 77 v5.0
Fortran 77	g77 version 2.95.2 19991024 (front end version 0.5.25)
Fortran 90	DIGITAL Fortran 90 v5.0

Table 2: Compilers used for generating binaries.

This interleaved execution of both analysis poses no problem for our code and data compaction needs, as the original constant propagation was already performed several times, interleaved with various other optimizations and analysis, such as useless code elimination, inlining, copy propagation, etc. The optimizations are repeated because they create optimization possibilities for each other that cannot be exploited by a single run over the optimizations.

6. CODE COMPACTION INTERACTIONS

Apart from the space benefits of dead data elimination, the primary effect of analyses described above is in the improvement of control flow analyses in the program. There are two sources for such improvements: first, these analyses allow us to resolve indirect control transfers more accurately, which in turn makes the control flow graph more precise and thereby improves the effects of dataflow analysis; and second, the elimination of pointers into the code from the data area, e.g., from within jump tables and virtual function tables, allows more code to be identified as unreachable and discarded. Both of these, in turn, have a beneficial effect on data elimination: improvements in the precision of the control flow graph lead to better constant propagation, while elimination of unreachable code eliminates load instructions that access memory, and thereby allow more data to be identified as dead and eliminated. Space constraints preclude a more detailed discussion of these interactions, but the interested reader is referred to [4].

7. EXPERIMENTAL RESULTS

For evaluating these algorithms, we have implemented them in SQUEEZE [4], 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 [13], a link-time optimizer oriented at speeding up programs.

The benchmark programs we used for evaluating our algorithms consist of all C-programs from the SPECint2000 benchmark suite, 252.eon, a C++ program from the SPECint2000 benchmark suite, five smaller C-programs from the MediaBench that are typical for embedded applications, and finally some programs of the SPECfp2000 benchmark suite: 168.wupwise, a Fortran 77 program, and 178.galgel, a Fortran90 program.

	base		code compacted		code and data compaction	
program	text	binary	text	binary	text	binary
164.gzip	59412	327760	35792 (60.2%)	254032 (77.5%)	34848 (58.7%)	237360 (72.4%)
175.vpr	107000	637056	72624 (67.9%)	514176 (80.7%)	71264 (66.6%)	483456 (75.9%)
176.gcc	434744	2262816	312688 (71.9%)	1795872 (79.4%)	312048 (71.7%)	1699264 (71.1%)
181.mcf	64072	345216	40192 (62.7%)	271488 (78.6%)	39024 (60.9%)	242400 (70.2%)
186.crafty	112684	635696	79600 (70.6%)	521008 (82.0%)	79280 (70.4%)	482784 (76.0%)
197.parser	92156	493232	59344 (64.4%)	378544 (76.7%)	58224 (61.2%)	347168 (70.4%)
253.perlbmk	221928	1144512	153616 (69.2%)	882368 (77.1%)	153280 (69.1%)	828192 (72.4%)
254.gap	216984	1025616	151200 (69.7%)	779856 (76.0%)	150176 (69.2%)	755136 (73.6%)
255.vortex	211320	1289600	126304 (59.7%)	961920 (74.6%)	125344 (59.3%)	895808 (69.5%)
256.bzip2	55288	311472	33424 (60.5%)	245936 (79.0%)	32432 (58.7%)	229376 (73.6%)
300.twolf	134556	736080	93872 (69.8%)	588624 (80.0%)	92720 (68.9%)	521712 (70.9%)
MEAN			*66.1%*	*78.3%*	*65.0%*	*72.4%*
adpcm	44560	257424	25328 (56.8%)	191888 (74.5%)	24272 (54.5%)	183520 (71.3%)
epic	71432	388960	44608 (62.5%)	298848 (76.8%)	43408 (60.8%)	277104 (71.2%)
gsm	63828	351712	38400 (60.2%)	269762 (76.7%)	37360 (58.5%)	249024 (70.8%)
mpeg2dec	68384	384976	43424 (63.5%)	303056 (78.8%)	42320 (61.9%)	281920 (73.2%)
mpeg2enc	85236	475168	57616 (67.6%)	376864 (79.3%)	56416 (66.2%)	347088 (73.0%)
MEAN			*62.1%*	*77.2%*	*60.4%*	*71.9%*
252.eon	178608	961136	91648 (*51.3%*)	625264 (*65.1%*)	86192 (*48.3%*)	520880 (*54.2%*)
168.wupwise	161440	824400	95376 (*59.1%*)	578640 (*70.2%*)	87664 (*54.3%*)	481952 (*58.5%*)
178.galgel	209868	1035424	133648 (*63.7%*)	748704 (*72.3%*)	125872 (*60.0%*)	658576 (*63.6%*)

Table 3: Number of instructions and binary program size (bytes) for the benchmarks generated by the Compaq compilers (base), after code compaction and after combined code and data compaction. The ratio's given are all compared to the base binaries.

The compilers we used to generate the binaries are given in Table 2. These compilers use different libraries, which is useful to show the generality of our techniques. All binaries were compiled with the -O2 flags, resulting in base binaries that are optimized for space and time. For linking, Compaq's ld was used with flags -r -d -z -m -non_shared. This way statically linked executables are produced, containing symbol and relocation information. The -m flag makes the linker 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 Tables 3 and 4 for binaries generated by Compaq and Gnu compilers. The average program size reductions for the SPECint2000 benchmarks are 27.6% and 32.1%, depending on the compilers used and therefore on the libraries linked with the program. Compared to the numbers for code compaction only, they are 5.9% and 5.5% higher. This results largely from the removal of dead data and less from additional elimination of code, as the gain in code size reduction is much smaller. The results for the MediaBench programs are similar.

The results for the C++ program, 252.eon, are quite remarkable. More than half of the instructions is removed from the program, which, together with the removal of dead data, results in a program compaction of 46.8%. The result is that the statically linked, compacted binary is 5.1% smaller than the dynamically linked one! The reason is the dynamically linked program consists for a large part of a dynamic string and symbol table.

The results obtained for 168.wupwise and 178.galgel show that also for scientific applications program compaction yields good results. Note that, despite the fact that the g77-compiled binary for 168.wupwise is more than a factor 2 smaller than the f77-compiled one (which is due to the use of much smaller libraries), the relative compaction results for both binaries do not differ that much. On the one hand, this confirms our believe that the size of a program is not only correlated to the functionality needed by the programmer, but also highly depends on the libraries used. On the other hand the size of the compacted binaries shows that there is much room for progression, as the f77-compiled and compacted binary is still more than a factor 1.8 larger than the g77-compiled and compacted one. The number of instructions in both binaries even differs with more than a factor of 2.

Table 5 compares the execution times for the base programs, the base programs with profile-directed code layout added, and the programs resulting from SQUEEZE. The experiments were run on a 500 MHz Compaq Alpha 21164 EV56 processor with a split primary direct mapped cache (8 KB each of instruction and data cache), 96 KB of on-chip secondary cache, 8 MB of off-chip backup cache, and 512 Mbytes of main memory, running Tru64 Unix 5.0a. 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 the compacted programs are, on the average, about 5% faster than the original programs.

Table 6 shows the total memory footprint (i.e. the largest amount of memory an application takes during its execution) for the MediaBench programs. The average compaction is 17.2%. This is not only due to the code and data compaction, but also to the removal of unnecessary stack-spills by 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 [5, 6, 7, 8, 11, 12]. 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 [5, 6, 7, 8]—which can be problematic for limited-memory devices—or require special

	base		code compacted		code and data compaction	
program	text	binary	text	binary	text	binary
164.gzip	57592	318592	30464 (52.8%)	228480 (71.7%)	29472 (51.2%)	211888 (66.5%)
175.vpr	100108	542544	62912 (62.8%)	411472 (75.8%)	61584 (61.5%)	380336 (70.1%)
176.gcc	434376	2139184	281040 (64.7%)	1557552 (72.8%)	280416 (64.6%)	1445952 (67.6%)
181.mcf	60252	326848	37040 (61.5%)	253120 (77.4%)	35872 (59.5%)	232400 (71.1%)
186.crafty	106204	574224	71008 (66.9%)	451344 (78.6%)	69872 (65.8%)	413008 (71.9%)
197.parser	86904	456608	53408 (61.5%)	341920 (74.9%)	52032 (59.9%)	310496 (68.0%)
253.perlbmk	210244	1085136	130816 (62.2%)	790224 (72.8%)	130912 (62.3%)	719344 (66.2%)
254.gap	186188	876944	115216 (61.9%)	614800 (70.1%)	114176 (61.3%)	590064 (67.3%)
255.vortex	213876	1112144	116400 (54.4%)	735312 (66.1%)	115280 (53.9%)	672160 (60.4%)
256.bzip2	49932	284528	28400 (56.9%)	210800 (74.1%)	27408 (54.9%)	202432 (71.1%)
300.twolf	123856	631984	77248 (62.4%)	459952 (72.8%)	76160 (61.5%)	420880 (66.6%)
MEAN			*60.7%*	*73.4%*	*59.7%*	*67.9%*
adpcm	41208	240848	22704 (55.1%)	183552 (76.2%)	21600 (52.4%)	166944 (69.3%)
epic	67196	368496	41040 (61.1%)	278384 (75.5%)	39888 (59.4%)	264752 (71.8%)
gsm	59180	328432	32800 (55.4%)	246512 (75.1%)	31680 (53.5%)	229888 (70.0%)
mpeg2dec	63064	363104	37424 (59.4%)	272992 (75.2%)	36288 (57.5%)	251664 (69.3%)
mpeg2enc	81420	444640	52800 (64.8%)	346336 (77.9%)	51584 (63.4%)	316576 (71.2%)
MEAN			*59.2%*	*76.0%*	*57.2%*	*70.3%*
168.wupwise	69784	395216	41024 (*58.8%*)	305104 (*77.2%*)	39008 (*55.9%*)	258416 (*65.4%*)

Table 4: Number of instructions and binary program size in bytes for the benchmarks generated by the GNU compilers (base), after code compaction and after combined code and data compaction. The ratio's given are all compared to the base binaries.

hardware support for executing the compressed code directly [11, 12]. 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 [1, 3, 9, 18]. 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%. 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 inter-procedural compiler optimizations aimed at eliminating code, can achieve significant reductions in code size, averaging around 30% [4]. 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 overall size reductions that are about 5-6% higher than that reported in our earlier work [4], this improvement coming mainly from the removal of dead data.

The elimination of unused data from a program has been considered by Srivastava and Wall [15] and Sweeney and Tip [16]. 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. 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 27–32% overall.

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. 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: a more precise analysis of stack behavior can lower the number of program points at which worst-case assumptions have to be made. Instead of not following the use of a stack-saved address, it will then be possible to follow its use from the places where the address is reloaded from the stack. Using a poly-variant partial evaluation for each produced address will produce better results as well.

Another way to increase the performance of these algorithms is to split the data blocks in smaller ones. At link-time, interval analysis could be a useful algorithm to head in this direction.

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 ineffeciencies though.

program	Compaq compilers			GNU compilers	
	base	profiled	compacted	base	compacted
164.gzip	1152	1111 (96.4%)	1155 (100.3%)	1180	1110 (94.3%)
175.vpr	919	897 (97.6%)	767 (83.5%)	1012	830 (82.0%)
176.gcc	865	813 (94.0%)	837 (96.8%)	874	874 (100.1%)
181.mcf	1463	1455 (99.5%)	1485 (101.5%)	1493	1476 (98.6%)
186.crafty	660	610 (92.4%)	577 (87.4%)	632	644 (102.6%)
197.parser	1800	1663 (92.4%)	1740 (96.7%)	1795	1724 (96.3%)
253.perlbmk	942	904 (96.0%)	872 (92.6%)	969	889 (92.3%)
254.gap	1008	956 (94.8%)	1053 (104.5%)	902	875 (97.0%)
255.vortex	1299	1202 (92.5%)	1023 (78.8%)	1603	1186 (74.4%)
256.bzip2	1139	1089 (95.6%)	1086 (95.3%)	1205	1023 (84.1%)
300.twolf	1657	1827 (110.3%)	1560 (94.1%)	1921	1750 (91.6%)
GEOM. MEAN	1173	1139 (*97.1%*)	1105 (*94.2%*)	1235	1126 (*91.1%*)
adpcm	11.5	11.7 (101.7%)	12.3 (107.0%)	15.1	15.2 (100.7%)
epic	11.6	11.3 (97.4%)	12.0 (103.4%)	14.1	16.7 (118.4%)
gsm	11.9	12.9 (108.4%)	11.8 (99.2%)	14.3	12.8 (89.5%)
mpeg2dec	11.5	10.8 (93.9%)	14.2 (123.5%)	21.2	19.3 (91.0%)
mpeg2enc	11.7	9.2 (78.6%)	11.5 (98.3%)	17.3	16.1 (93.1%)
GEOM. MEAN	11.6	11.2 (*96.5%*)	12.4 (*106.2%*)	16.4	16.0 (*97.7%*)
252.eon	780	792 (101.5%)	848 (108.7%)	-	-
168.wupwise	1082	1114 (103.0%)	1013 (93.6%)	1255	1213 (96.7%)
178.galgel	2697	2827 (104.8%)	2728 (101.1%)	-	-

Table 5: Execution times for the base binaries, the profile-feedback generated binaries and the code and data compacted binaries.

program	base	compacted
adpcm	312 K	208 K (66.7%)
gsm	456 K	344 K (75.4%)
epic	1.70 M	1.58 M (92.9%)
mpeg2dec	888 K	768 K (86.5%)
mpeg2enc	1.88 M	1.74 M (92.6%)
MEAN		*82.8%*

Table 6: Total Memory Footprint for the MediaBench programs.

10. REFERENCES

[1] 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.

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

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

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

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

[6] 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.

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

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

[9] 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.

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

[11] 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.

[12] K. D. Kissell. Mips16: High-density mips for the embedded market. In *Proc. Real Time Systems '97 (RTS97)*, 1997.

[13] R. Muth, S. Debray, S. Watterson, and K. De Bosschere. alto : A link-time optimizer for the compaq alpha. *Software Practice and Experience*, 2001. (to appear).

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

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

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

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

[18] M. J. Zastre. Compacting object code via parameterized procedural abstraction. Master's thesis, Dept. of Computing Science, Univ. of Victoria, 1993.