IDENTIFICATION AND REMOVAL OF PROGRAM SLICE CRITERIA FOR CODE SIZE REDUCTION IN EMBEDDED SYSTEMS

Mark Panahi, Trevor Harmon, Juan A. Colmenares*, Shruti Gorappa, and Raymond Klefstad
Department of Electrical Engineering and Computer Science
University of California, Irvine, CA 92697, USA
{mpanahi, tharmon, jcolmena, sgorappa, klefstad}@uci.edu

Abstract: Code shrinking is a technique for reducing the size of a software system by detecting and eliminating unused features. It is a static approach that is limited to information known at build time. We demonstrate how to extend code shrinking to take dynamic behavior into account. Our technique, Slice Criteria Identification and REmoval (SCIRE), combines conditional compilation and code shrinking techniques with information provided by the developer about run-time behavior. This integration allows SCIRE to achieve code reduction that is close to optimal while remaining mostly automatic. Our results show that SCIRE reduces code size substantially: In many cases, removal of unused features shrinks middleware footprint to less than 5% of its original size, allowing CORBA and other large software libraries to be used in embedded systems.

1. STATIC FOOTPRINT MINIMIZATION TECHNIQUES

Over the years, three basic techniques have emerged for reducing the size of executable code ("footprint"):

1. **Specification subsetting**: Removal of infrequently used features from a standard shared library, API, or class hierarchy.
2. **Conditional compilation**: Manual removal of features on a per-application basis using preprocessor directives.

*Also with the Applied Computing Institute, Faculty of Engineering, University of Zulia.

Panahi, M., Harmon, T., Colmenares, J.A., Gorappa, S., Klefstad, R, 2007, in IFIP International Federation for Information Processing, Volume 231, Embedded System Design: Topics, Techniques and Trends, eds. A. Rettberg, Zanella, M., Dömer, R., Gerstlauer, A., Rammig, F., (Boston: Springer), pp. 269–278.

3. **Code shrinking**: Automatic removal of dead code by pruning a graph of all possible execution paths.

Each approach offers a unique set of strengths and weaknesses, and therefore any innovation beyond these existing techniques requires a careful comparison to discover avenues of improvement. Toward that end, we have identified three criteria for evaluating each approach:

- **Flexibility** *(Criterion 1).* A flexible approach gives the developer freedom to choose the features necessary to accommodate design requirements.
- **Maintainability** *(Criterion 2).* A maintainable approach is one that remains manageable as the size of the application grows. Meeting this criterion requires some form of automation to avoid loss of productivity as the code base increases.
- **Run-time-awareness** *(Criterion 3).* A "run-time-aware" approach takes into account knowledge of application-specific requirements and explicitly removes features that will (according to the developer) never be required at run-time.

For the remainder of this section, we examine the three approaches to footprint reduction from an application developer's standpoint and discuss if and how they meet the three criteria listed above.

Specification Subsetting. With this approach, software architects specify reduced profiles (e.g., an API or class hierarchy) of existing software libraries with an eye toward resource-constrained embedded devices. These reductions are obtained by removing features from standard library specifications. This task necessarily implies a trade-off between the conflicting goals of small footprint and usability. Thus, the memory footprint reduction obtained from specification subsetting varies widely and depends heavily on the original profiles and the objectives established by the specification.

A well-known example is the Java 2 Platform, Micro Edition (J2ME). By removing large portions of the standard Java class library and providing a simplified virtual machine interpreter, J2ME meets the stringent requirements of many embedded systems. However, J2ME's main disadvantage is that the reduced profiles are defined beforehand, preventing adaptation to unanticipated application requirements *(Criterion 1).*

Moreover, specification subsetting does not take advantage of the knowledge of a system's run-time conditions *(Criterion 3).* In practice, features in reduced profiles are considered fixed, and even if an application developer can guarantee that certain features will never be used during execution, such features cannot be removed.

Conditional Compilation. Conditional compilation involves a preprocessor examining compile-time configuration flags to decide which portions of a shared library belong in the final executable package. It is most often applied in C/C++ using the `#ifdef` directive to allow retargeting of libraries to different platforms.

Because conditional compilation is applied on a per-application basis, developers can include only the features they need. It imposes no restrictions on accommodating application requirements *(Criterion 1)* and allows the developer to exploit knowledge of the system's run-time conditions *(Criterion 3)*. Thus, in theory, conditional compilation can reduce the footprint of a program to the absolute minimum. In practice, however, obtaining the theoretical minimum code size requires enormous effort, and therefore developers routinely accept sub-optimal results when using this technique.

Code Shrinking. Many compilers include optimization algorithms to reduce the footprint of a program. These algorithms typically eliminate redundant and unused code based on an *intra*-procedural analysis. However, they can easily fail to identify unused code due to a lack of information necessary for a full *inter*-procedural analysis. When parsing the shared library of an API, for example, compilers have no way of knowing which of the API's functions will (or will not) be called by a given application.

For inter-procedural elimination, there exist footprint optimization tools that perform whole-program analysis. These optimizers, also known as *code shrinkers*, reduce footprint by analyzing an entire program, including any shared libraries that it uses, and removing unnecessary ("dead") portions of code. Code shrinkers are able to detect this dead code by building a call graph of all possible execution paths of a given program. Any classes and methods not in the graph are removed, as shown in Figures 1 and 2. This memory reduction process is called *code shrinking* or *dead code elimination.*

```
public class Hello {
  public Hello(boolean b) {
    if (b) methodA();
    else methodB();
  }
  private void methodA() {
    System.out.println("Method A");
  }
  private void methodB() {
    System.out.println("Method B");
  }
  private void methodC() {
    System.out.println("Method C");
  }
  public static void main(String[] args) {
    Hello h = new Hello(args.length > 0);
  }
}
```

Figure 1. Java compilers generate code for method C, but code shrinkers correctly identify it as dead code and remove it.

A code shrinker's ability to remove unused features from a program is effective but still sub-optimal. Because it has no knowledge of run-time conditions, features may appear to the shrinker as having structural

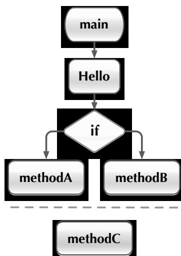

Figure 2. Code shrinkers build call graphs, such as this graph of Figure 1, to conclude that method C is never called and can be removed.

Technique	**Flexible** *(Crit. 1)*	**Maintainable** *(Crit. 2)*	**Run-time-aware** *(Crit. 3)*
Specification subsetting		✓	
Conditional compilation	✓		✓
Code shrinking	✓	✓	

Figure 3. A comparison of footprint reduction techniques.

dependencies even though they are independent according to the application's true run-time behavior. In other words, code shrinkers will not remove any code that could *potentially* be executed at run-time *(Criterion 3).*

Qualitative Comparison. Table 3 summarizes the three footprint reduction techniques according to our criteria. Note that none of these techniques is able to meet all criteria. We observe, however, that if the code shrinking approach could be made run-time aware, it would meet all three criteria. This observation is the basis for our SCIRE approach, as described in the following section.

2. SCIRE

SCIRE, *Slice Criteria Identification and REmoval*, is our technique for enhancing the ability of code shrinkers to discover dead code by incorporating knowledge of run-time behavior. We first explain SCIRE's foundation in program slicing [Weiser, 1981] and then how it may be applied to the problem of footprint reduction.

Program Slicing. Program slicing is a well-known software analysis technique that is used to identify code that influences or is influenced by a point of interest in a program. The point of interest is called the *slice criterion* and could be a variable or any program statement. The code that is affected by the criterion is known as the *forward slice*, and the code that affects the criterion is known as the *backward slice.*

Code shrinking tools mainly use the forward slicing technique to remove unused code. However, code shrinking tools are based only on *static program slicing*, where the slice is determined based on the static program structure and does not include any run-time information. In

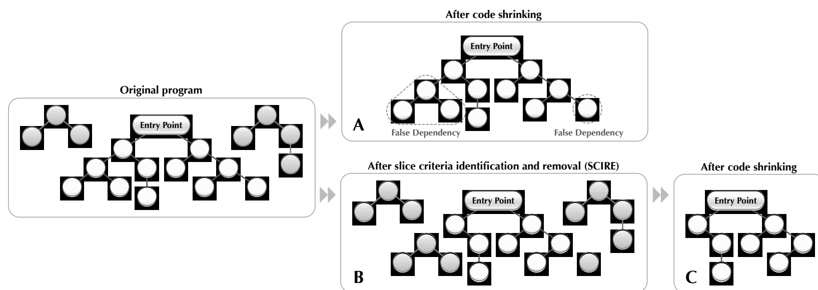

Figure 4. Removing syntactic dependencies (that is, the slice criteria) between features can enhance the effect of code shrinking.

contrast, *dynamic program slicing* [Agrawal and Horgan, 1990] leverages run-time information—in addition to static structure—to identify the true program slice for a given execution. Dynamic slices are therefore smaller in most cases and no larger than their corresponding static slices. Their precision varies depending on the slicing approach [Agrawal and Horgan, 1990, Zhang et al., 2003]. In our approach, we rely on the developer's run-time knowledge of a given slice criterion to apply dynamic slicing. As a result, only the code truly necessary for program execution is retained in the final footprint.

Application of Program Slicing to Footprint Reduction. Program slicing techniques have traditionally been used to examine and understand program execution, for debugging, and for testing. In this paper, we present a new application of this technique to achieve code footprint reduction. To the best of our knowledge, dynamic program slicing has not previously been used to perform code reduction.

Our approach, Slice Criteria Identification and REmoval (SCIRE), is simple in that it only requires the identification of program slice criteria, rather than the entire program slice associated with a feature, as with a purely conditional compilation (PCC) approach. Through the identification and optional removal of such dependencies, SCIRE enhances the code shrinking approach. The key observation is that certain slice criteria, such as an `if` statement, may mark the start of a program slice such that the decision to execute these slices cannot be made until run-time.

Figure 4 presents a generalized diagram of this process. Each circle, or node, represents a basic program block. The links between nodes represent dependencies between the blocks, such as `if` statements or function calls. Analysis begins at the node marked *Entry Point*, usually the standard `main` function. The shaded nodes are program blocks that

have no dependency on the entry point and can thus be removed through code shrinking, as shown in part A. The nodes labeled *False Dependency* are reachable from the entry point, but they are not actually needed by this application at run-time.

In our alternative process, beginning in part B, SCIRE is applied to remove the slice criteria binding the false dependencies to the entry point. When code shrinking is subsequently applied (part C), it is able to remove both the truly independent program slices and the falsely dependent program slices. As more slice criteria are identified and removed, the cumulative effect on the overall footprint can be significant, typically around 30-50% more than code shrinking alone, according to our experimental results.

3. IDENTIFYING PROGRAM SLICE CRITERIA

The effectiveness of SCIRE depends on our ability to identify slice criteria that bind unnecessary features in software libraries to a given application. Finding many of the features in libraries, such as middleware and foundation classes, is difficult because code associated with these features is typically scattered throughout several locations. In addition, the codebase may be very large and complex, making manual identification of slice criteria extremely difficult and time-consuming.

To overcome these drawbacks, we have developed our own tool called Shiv for SCIRE that is based not on aspects but on static program slicing. This change allows Shiv to focus on SCIRE's overall goal of footprint reduction. By taking into account the code size of each program slice, it can provide a visual representation of slices along with their corresponding contribution to the overall footprint. The developer can then use this visualization to more easily identify large program slices as candidates for removal.

Shiv offers the developer two choices for visualizing the call graph: a tree or a treemap. The tree is a traditional node-link visualization, where the root of the tree is the chosen slice criterion, each node is a method in the program, and each link is a dependency from one method to another (i.e., a method call). The treemap alternative represents the same tree structure, but it does so using a space-filling algorithm [Johnson and Shneiderman, 1991] that takes into account the code size of each method in the call graph. This allows the largest program slices to be identified easily.

We illustrate the benefits of Shiv using the canonical bubblesort algorithm as an example. When Shiv is provided the program in Fig-

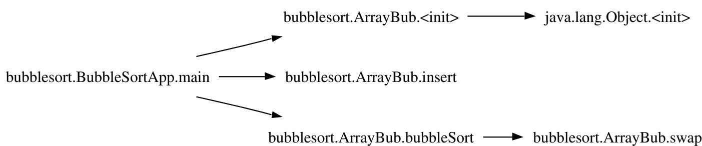

Figure 6. A node-link tree visualization generated by Shiv of the program in Figure 5.

ure 5 as input, it generates the tree shown in Figure 6. These figures show the relationship between a program's source code and its visualization in Shiv. For example, the `main` method calls `ArrayBub.insert`, `ArrayBub.bubbleSort`, and `ArrayBub`'s constructor, and therefore each of these calls appears as a child of `main`.

```
class ArrayBub {
  private long[] a; private int nElems;
  public ArrayBub(int max) {
    a = new long[max];
    nElems = 0;
  }
  public void insert(long value) {
    a[nElems] = value;
    nElems++;
  }
  public void bubbleSort() {
    for (int out = nElems-1; out > 1; out--)
      for (int in = 0; in < out; in++)
        if ( a[in] > a[in+1] )
          swap(in, in+1);
  }
  private void swap(int one, int two) {
    long temp = a[one];
    a[one] = a[two];
    a[two] = temp;
  }
}
class BubbleSortApp {
  public static void main(String[] args) {
    ArrayBub arr = new ArrayBub(100);
    arr.insert(77); arr.insert(99); arr.insert(44);
    arr.bubbleSort();
  }
}
```

Figure 5. This simple bubblesort program demonstrates the effectiveness of our slice criteria identification tool, Shiv. Visualizations of this program can be seen in Figures 6, 7, and 8.

The true importance of Shiv can be seen if a slight change is made to the Figure 5 code. If a single `println` statement is added, the call graph explodes into the tree shown in Figure 7. The Shiv output reveals that the single slice criterion of the `println` statement is the source of an extremely large program slice. (In this case, the large size is due to the security checks that Java requires for all program output.) We can also see possible features starting to emerge, exhibited by clusters in the tree. These clusters indicate program slices that could be modularized or perhaps removed entirely, depending on the true run-time behavior of the program using the slice.

The tree in Figure 7 is informative, but recall that the motivation of Shiv, and our SCIRE technique in general, is footprint reduction. For such a goal, a simple node-link diagram does not capture information about code size. For example, a collection of very small

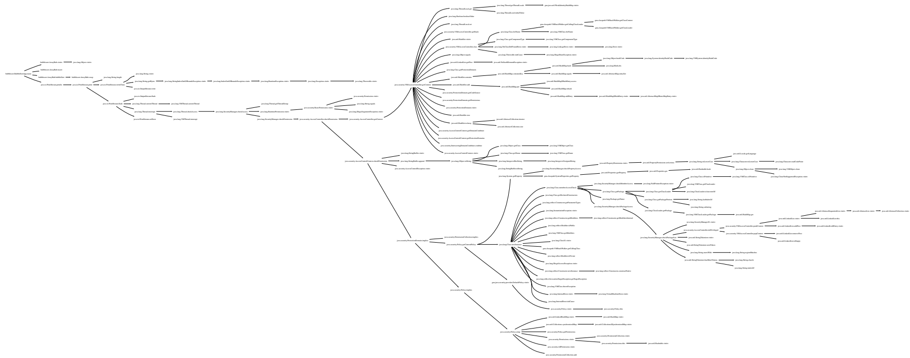

Figure 7. A node-link tree visualization generated by Shiv after adding a `println` statement to the program in Figure 5. Although the labels are not legible here, the structure shows that, in comparison with Figure 6, the size of the call graph has exploded with the addition of a single line of code.

methods may appear in the tree as a large cluster, while a single large method that takes up even more space would appear only as a single node. This potential to mislead the developer compelled us to offer an alternative visualization in Shiv: the treemap.

Using the Treemap 4.0 program [Baehrecke et al., 2004], Shiv produces a visualization of the program slice that quantifies the largest methods contributing to that slice. For example, Figure 8 shows the same bubble-sort program of Figure 5, again with an added `println` statement. By examining the branch labels of this squarified treemap, we can see that the largest contributors to code size include the `RuntimePersmission` constructor and the `HashMap.put` method, for they are represented by the largest blocks. Looking further up the tree, we note that the `check-Access` method corresponds to an extremely large block. Thus, security checks make up the vast majority of the static footprint of this program.

Based on this information, the developer can select slice criteria that, if removed, would have the largest impact on footprint reduction. Note that the act of actually removing a program slice requires an explicit action on the part of the developer (for instance, using the techniques in Section 2) and cannot be achieved by Shiv alone. However, for complex real-world programs with many dependencies, Shiv greatly simplifies the overall SCIRE technique. It prevents the developer from having to examine a large codebase method-by-method, looking for slice criteria. In addition, it can provide insights that may not be readily visible, such as the fact that the `println` statement has a dependency on Java's security module.

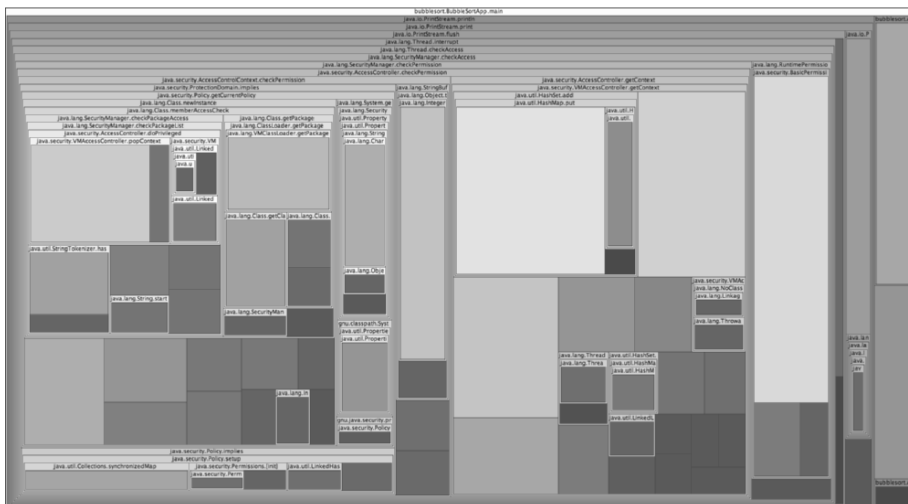

Figure 8. Shiv generates visualizations of embedded systems software, such as this treemap of the program in Figure 5. Each block represents the static code size of a program method: Larger, brighter blocks correspond to larger method sizes. Such visualizations help the developer easily identify methods that adversely affect code size.

4. EXPERIMENTAL RESULTS

This section presents the results of applying the SCIRE approach in conjunction with ProGuard,[1] a Java code shrinker, for reducing the static code size of JacORB 2.2,[2] a Java implementation of CORBA. For our measurements, the J2ME Connected Limited Device Configuration (CLDC) gives a reasonable target for code size: devices that have 160-512 KB of memory available for the Java platform and applications. Because the target size for these embedded devices is so small, even modest reductions in footprint become significant.

Our footprint measurements were based on the functionality required by two sample applications: 1) a *Simple Remote Client* and 2) *Supplier Dispatch using FACET*[3]. We collected the following measurements for each application: 1) the original size of JacORB before any reduction has taken place, 2) the size of the resulting library after code shrinking,

[1] http://proguard.sourceforge.net/
[2] http://www.jacorb.org/
[3] http://www.cs.wustl.edu/~doc/RandD/PCES/facet/

and 3) the size of the resulting library after applying both SCIRE and code shrinking.

Figure 9 shows that code shrinking alone provides a substantial reduction of code size: over 90% in relation to the original library. However, our proposed approach—the joint application of SCIRE and code shrinking—provides additional reduction: about 96% from the original size, and over 50% beyond the application of code shrinking alone. This additional reduction shows the effects of dependencies on unused features that remained in the code. Only the application of the SCIRE approach can enable the removal of such features. In fact, for both applications, SCIRE is necessary to bring the library size within the range specified by the CLDC.

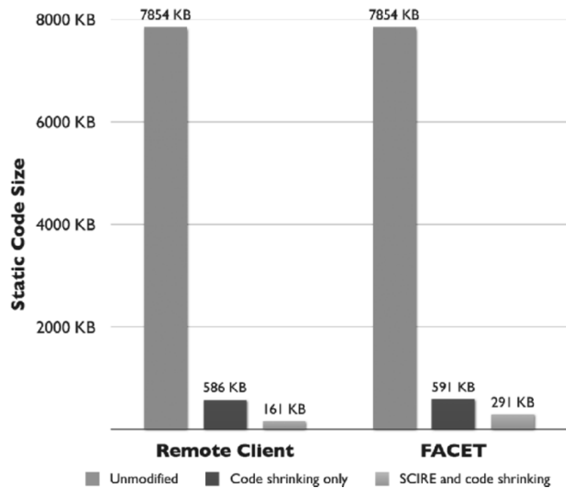

Figure 9. Results of footprint reduction in two sample applications: a CORBA remote client and a FACET event channel client.

REFERENCES

[Agrawal and Horgan, 1990] Agrawal, H. and Horgan, J. R. (1990). Dynamic program slicing. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, volume 25, pages 246–256, White Plains, NY.

[Baehrecke et al., 2004] Baehrecke, E. H., Dang, N., Babaria, K., and Shneiderman, B. (2004). Visualization and analysis of microarray and gene ontology data with treemaps. *BMC Bioinformatics*, 5(84).

[Johnson and Shneiderman, 1991] Johnson, B. and Shneiderman, B. (1991). Treemaps: a space-filling approach to the visualization of hierarchical information structures. In *VIS '91: Proceedings of the 2nd conference on Visualization*, pages 284–291, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Weiser, 1981] Weiser, M. (1981). Program slicing. In *ICSE '81: Proceedings of the 5th international conference on Software engineering*, pages 439–449, Piscataway, NJ, USA. IEEE Press.

[Zhang et al., 2003] Zhang, X., Gupta, R., and Zhang, Y. (2003). Precise dynamic slicing algorithms.