Honours Year Project Report

Support for PROGMEM in embedded systems

By

Naomi Leow Wen Xin

Department of Computer Science

School of Computing

National University of Singapore

2016

Honours Year Project Report

Support for PROGMEM in embedded systems

By

Naomi Leow Wen Xin

Department of Computer Science

School of Computing

National University of Singapore

2016

Project No: H002970
Advisor: Associate Professor Hugh Anderson
Deliverables:
Report: 1 Volume

Abstract

The Arduino is a microcontroller which operates on a modified Harvard Architecture. It has 3 types of memory: *Flash Memory*, where the loaded program's code is stored, *SRAM*, where data for the executing program is stored and the *EEPROM*. On the Arduino Uno, there is only 2KB of SRAM as compared to 32KB of Flash.

As the amount of SRAM available on the Arduino is limited, loading all of the program's data into the SRAM may result in the program eventually running out of available SRAM. One way of reducing a program's SRAM usage is to keep constant data in Flash memory, loading them into SRAM when they are needed, instead of loading them all into the SRAM at start up. The *pgmspace.h* Library allows data bound to variables declared with the keyword PROGMEM to be stored in Flash memory. However, this requires additional code to load data into SRAM before they can be used by the program being executed, which introduces room for error. Programmers will have to aware of whether the pointers they are working with point to a location in SRAM or Flash and handle them differently.

The goal of this project is to reduce the amount of SRAM required for the execution of an Arduino program, by making use of the Flash memory and the *pgmspace.h* memory. The focus is on reducing the RAM footprint of the program reducing the amount of memory required by unmodified Strings at runtime. As proof of concept, a tool has been developed using the ROSE compiler framework to transform a given code into a semantically equivalent form that requires less memory.

Subject Descriptors:

- Memory Optimisation
  - Dataflow Analysis
    - Def-use Analysis
    - Alias Analysis
  - Program Transformation
    - ROSE Source-to-Source Translator

Implementation Software and Hardware:

Docker, Ubuntu 14.04, GCC-4.8

Acknowledgement

I would like to thank my advisor, Prof Hugh Anderson for his advice during the course of the project.

Special thanks to friends and family for their support.

Also, special mention to Alex Marginean for putting up his scripts for setting up the ROSE compiler in a Docker container.

Table of Contents

Chapter 1

Introduction

1.1 Overview

The Arduino is an open-source prototyping platform that consists of a programmable circuit board with a microcontroller as its computing unit [1]. While the exact specifications of the microcontroller vary across the different models of Arduino boards available, most of them use ATMEL AVR micontrollers. The AVR microcontrollers have a modified Harvard architecture, where instructions and data are stored separately but there are special instrunctions to store or retrieve data from program space. The Arduino Due uses an ATMEL SAM microcontroller, but has a modified Harvard architecture like the rest of the Arduino boards.

1.1.1 Programming an Arduino

While code written in C or C++ can be compiled with gcc or g++ (avr-gcc or avr-g++ for AVR microcontrollers) to generate binaries which can be uploaded and run on the microcontroller, Arduino has an IDE which simplifies and hides

[1] http://www.arduino.cc/en/Guide/Introduction

the complexities of the commands needed to invoke the gcc based tool chain for this process.[2]

The Arduino IDE facilitates the development in a simplified subset of C++, which heavily relies on API developed by Wiring(Barrag'an, 2004)[3], which makes interfacing with the I/O pins on the board relatively simple. Code written in the IDE is referred to as a Sketch and have a .ino extension.

Sketches compiled and uploaded to the microcontroller via the IDE follows the following structure:

```
//library includes
void setup() {
//setup code
}
void loop() {
//code to run repeatedly
}
```

The *setup()* function is only run once at the start, thereafter the *loop()* function is run again and again unless a process termination signal is received.

1.1.2 Arduino Memory Architecture

With the exception of the Arduino Due, all Arduino boards have microcontrollers with 3 memory locations:

[2]http://www.arduino.cc/en/Reference/HomePage

[3]http://wiring.org.co/reference/

1. **Flash Memory**

   The flash memory is where the Arduino sketch is stored. The Arduino Uno has 32k bytes of flash memory available, with 0.5k bytes reserved for the bootloader. Boards which use the ATmega2560 have 256k bytes of flash, with 8k bytes reserved for the bootloader.

   Data associated with static variables are loaded into the .data section of the SRAM at start up. String literals used in the program are among the static data loaded into the .data section. The *pgmspace.h* library of AVR-libc allows variables marked with the PROGMEM directive to be left the flash at start up. However, such data must still be loaded into RAM before they can be used in the execution of the program.

2. **Static Random Access Memory**

   The amount of static random access memory (SRAM) available for use is very limited - only 2k bytes are available on boards such as Uno which use the ATmega328, while 8k bytes are available on boards that use the ATmega2560. With each character taking up 1 byte of memory, string literals are one major guzzler of SRAM memory.

3. **EEPROM**

   Like the flash memory, data stored in the EEPROM memory persist when power to the microcontroller is cut. Unlike the flash memory, non constant data which is generated at runtime can be stored in the EEPROM. However, only 1k byte and 4k byte of memory are available in the ATmega328 (used by the Uno) and the ATmega2560 respectively.

The Arduino Due has a similar architecture with regards to the Flash and SRAM memory. However, the Due has no EEPROM available. As a possible alternative to having an external EEPROM, ATMEL has an Embedded Flash service for the SAM microcontroller which allows data to be written to the flash memory at runtime. Memory in the Due is less scarce as compared to the rest of the boards, with 96k bytes of SRAM and 512k bytes of flash. *pgmspace.h* is not available for the SAM microcontroller in the Due, but Arduino has included a compatability header in its library for the SAM architecture, so code written with PROGMEM directives works as expected on the Due. For the Due, all static variables declared as *const* is left in flash at start up; string literals assigned to *const* pointers are also not loaded into RAM at start up.

1.1.3 Problem Overview

Given the limited amount of SRAM available on the AVR microcontrollers used by most of the Arduino boards, programs may run out of SRAM at run time since data must be loaded into SRAM before they can be used. This is especially pertinent for programs which work with strings, since each character takes up a byte of SRAM. Hence programs which work with Strings, such as Webservers or programs which interact with LCD displays, can very easily run out of SRAM at runtime.

The *pgmspace.h* library allows data to be placed in the .progmem.data segment of the flash memory at start up. However, additional instructions are needed to load data into SRAM before it can be used at runtime. This also introduces room

for error as pointers to data in Flash have to be handled differently.

There is a $F()$ macro [4] which casts the Flash memory address of the given string as a $\_\_FlashStringHelper$. The copy and concat methods of Arduino's String class as well as the print methods in Arduino's adaptation of the standard I/O library have been overloaded to work with the $\_\_FlashStringHelper$ cast, which serve as wrappers to the corresponding functions for handling strings stored in Flash memory in the *pgmspace.h* Library. However, strings wrapped with the F() macro can only be used as an function parameter. They cannot be used in variable assignment or declarations While convenient, using the String class to work with strings might not be a good idea as it uses memory allocated from the heap, which might lead to memory fragmentation in the SRAM, wasting precious SRAM space.

When the result of a string literal wrapped with the F() macro is used with the print methods, the given string literal is read into memory character by character and written directly to the output stream, so no more than 1 additional byte of memory is needed to print the wrapped string literal.

A downside to just casting string literals with the F() macro is string literals with exactly the same characters that occur multiple times in the code will also be stored as multiple instances of the same string literals in Flash. This takes up unnecessary space in flash.

[4] http://playground.arduino.cc/Learning/Memory

1.2 Goals

The goal of this project is to reduce the amount of SRAM required for the execution of an Arduino program, by making use of the Flash memory and the *pgmspace.h* memory. While leaving all static data in Flash and loading them into SRAM only when they are needed at runtime will likely significantly reduce the RAM footprint of the program at runtime, the focus will be on string literals, which can consume a significant proportion of SRAM.

A key objective is to develop a tool which can reduce the runtime SRAM usage of a program by reducing the amount of space taken up by string literals during runtime. A source-to-source transformation will be done on a given Arduino Sketch code file, producing another Arduino Sketch code file which implements some optimisations on string literals in the code.

1.2.1 Contributions

A tool which performs a source-to-source transformation has been developed using the ROSE framework, which is a framework for program analysis and source code transformation. It supports C and C++ code, hence it can be used to work with Arduino Sketch code, which is based on C++. Reduction of space needed for string literals is done in a two step transformation:

1. Merge all occurences of the same string literal.

2. Mark string literals meeting conditions described in section 3.3, leaving them in flash at start up.

1.3 Organisation of Report

Chapter 2 is a summary of the research done at the start of the project. Section 2.1 is a summary of related work with regards to reducing the RAM consumption of programs running on the Arduino and other embedded system architecture in general. Section 2.2 describes the requirements for building a source to source transformation tool and explains my rationale for using the ROSE framework, with a description of the ROSE compiler in section 2.2.2.

Chapter 3 describes the implementation of the tool. Analysis of the transformation done by the tool is described in chapter 4. A discussion of further work which can be done to improve the tool is described in Chapter 5.

Chapter 2

Background

2.1 Related Work

2.1.1 CComp: Offline RAM compression

In the paper, *Offline Compression for On-Chip RAM* (Cooprider & Regehr, 2007), Cooprider and Regehr described how they reduced the memory footprint of a program's data by variable compression - reducing the number of bits needed to represent the values referred to by each variable in memory. Based on the variable compression techniques they developed, they developed CComp, a tool which performs source to source transformations on C programs for embedded systems to reduce the RAM consumption of the programs.

For each variable in the program, an analysis is done to determine an estimate of the set of values which can be assigned to the variable at each stage of the program, considering all possible code paths. If the size of the estimated set is less 2^n, where n is the size of the variable in memory in bits, the variable can be compressed. An invertible function is defined mapping the set of possible values to a smaller set which can be represented by fewer bits can be defined. In cases

where such a function cannot be easily defined, the possible values of the variable are stored in a compression table in Flash memory. The values are then referenced by their indices in the table.

In addition, duplicate compression tables are eliminated so variables whose range of possible values are the same will refer to the same compression table.

On their test applications, the team achieved an average of 12% reduction in RAM usage through variable compression. Additional savings in RAM usage was also achieved by removing dead variables found in their analysis.

2.1.2 Flattening the call stack

In the work by Yang et al (Xuejun Yang, 2009), the size of a program's runtime call stack is reduced by transforming function calls and returns into jump operations. The transformations were applied to C code for microcontroller units. There are cases with scenarios known as flattening hazards in which a program cannot be completely flattened. For cases where the program is completely flattened, there will only be one stack frame throughout the program's runtime. Since there is only one function, the main function, global variables are lifted into the main function's scope. As intraprocedural optimisation can be done better than interprocedural optimisation, this allows the transformed code to be better further optimised by the compiler.

However, flattened code also poses additional challenges for the compiler as it creates apparent paths for the compiler's optimisers which turn out to be never traversed at runtime. To address the issue, the team modified the GCC compiler

to take into account the callgraph and local variable declarations from the original unflattened code in the optimisation of the flattened code.

The team's implementation of flattening saw an average reduction in RAM usage of 20% against their test programs. However, there was one instance where there was a slight increase in RAM usage after flattening due to register spilling as a result of the increased register pressure of the transformed code.

2.1.3 Related work on the Arduino

Flash by Mikal Hart

[1] The library is a *pgmspace* library wrapper. It supports storing strings and arrays and tables of the datatyles supported by *pgmspace* in the Flash memory.

It works by creating a wrapper class for strings, arrays and tables respectively. The methods and operators in the wrapper classes does the necessary type casting and loading of data from the Flash memory using the *pgmspace* library, thus abstracting the complexities of working with *pgmspace* from the user. However, the library is not exhausive and certain PROGMEM compatible methods such as *strcmp_P* are not implemented.

Due to the declaration of a virtual method in the base class for all wrapper classes, using the library will incur an additional overhead in SRAM usage as a vtable will have to be created for the virtual function in SRAM.

[1]Original Source: http://arduiniana.org/libraries/flash

Updated repo: https://github.com/hemantsangwan/Arduino-Flash/blob/master/Flash.h

2.2 Requirements and available tools

In order to perform the source-to-source transformation, program analysis will have to be done on the source code to gather information on what string literals are used in the code and where they are used. Before analysis can be performed, an abstract syntax tree (AST) of the source code must be constructed. Subsequent program transformations will also be performed on the AST.

Since the Arduino sketches are a subset of the C++ language, a C++ AST generator will be able to do the job. Ideally, the chosen tool should be able to $''unparse''$ the AST back to code as well, which removes the need to write a module to handle the generation of the transformed source code from the modified AST.

2.2.1 Clang and Libtooling

Clang is an open source compiler for the C family languages built over LLVM. It provides an interface through which one can build source code analysis tools that leverage on the AST generated by Clang. One possible interface for program analysis is the LibTooling library.[2]

By extending Clang's ASTFrontendAction, ASTConsumer and RecursiveAST-Visitor classes, one can develop a tool that traverses the AST built by Clang's frontend. However, Clang's AST is not mutable. As I would need to modify or annotate the AST in the course of my analysis, I would not be able to rely on the interface provided by clang for my tool. A deeper understanding of Clang's implementation is needed in order to make use of Clang's AST for my analysis.

[2]http://clang.llvm.org/docs/LibTooling.html

Clang does not provide a means by which an AST can be converted back to C++ source code. Nevertheless, it has a Rewriter, which allows one to generate a modified version of an existing source code by specifying a location at which to insert, remove or replace text. Hence, transformations of an existing source code can be done by writing an ASTVisitor which produces a modification of the source code using the rewriter and the source information it gains as it traverses the AST. However, the inability to directly generate C++ code from the AST makes Libtooling less robust for the implementation of my project.

I studied the possibility of using Clang's LiveVariables analysis tool to determine the live String literals at each statement of the original source program. However, Clang does not provide a Libtool interface for this tool, hence like the case of Clang's AST, actual usage of the LiveVariables tool would require a deeper analysis of the implementation of the tool.

2.2.2 The ROSE Compiler Framework

ROSE is a source-to-source compiler framework, first developed by (Daniel J. Quinlan & Kowarschik, 2003). It supports a number of languages including C and C++. To generate an AST of a given C or C++ source code, it uses the Edison Design Group(EDG) frontend[3] to generate an initial AST of the source code, which is also used by comercial C++ compilers such as the Intel C++ compiler [4]. It uses a modified version of the Sage++, known as SageIII to build an intermediate representation AST of the source code from the AST generated by the EDG frontend.

[3]https://www.edg.com/c

[4]https://software.intel.com/en-us/get-started-with-cpp-compiler-for-osx-parallel-studio-xe

Subsequent analysis and transformation is done on the Sage intermediate AST.

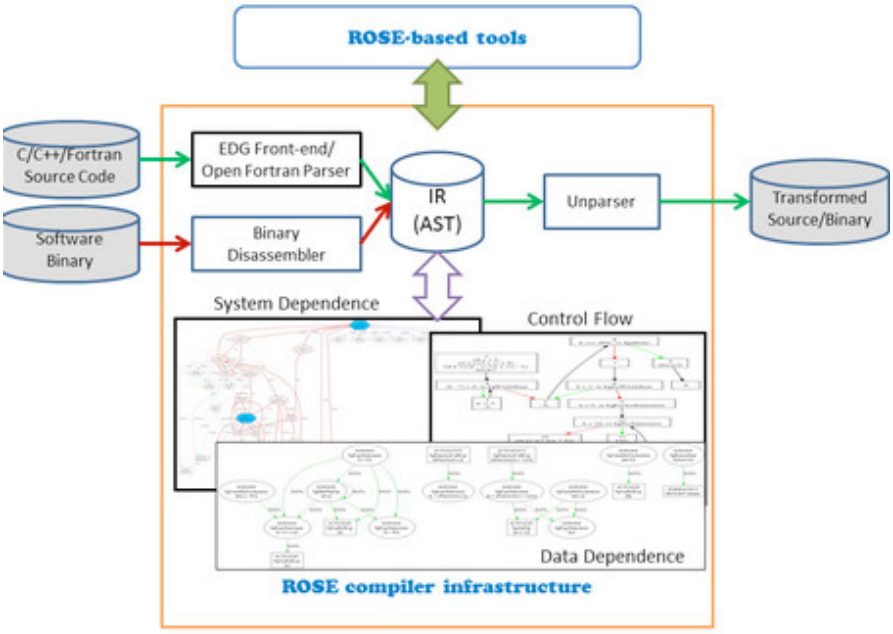


Figure 2.1: ROSE Framework Architecture, taken from (Quinlan & Liao, 2011)

ROSE provides an API for both bottom up and top down traversal of the Sage intermediate AST. It is also possible to annotate the AST with attributes, which can come in useful for program analysis.

ROSE also comes with a framework for Dataflow analysis. It generates a control flow graph (CFG) from the intermediate AST. Each node in the AST has a corresponding node in the CFG it is possible to skip over nodes during the traversal for the dataflow analysis by overidding the default filter.

The framework provides support for both forward propagation and backward propagation dataflow analysis. The lattice for each node for each analysis is stored in a global *NodeState* object so it can be easily accessible without traversing the

entire dataflow graph.

The framework also has a few commonly used anaysis, such as Def-use and Liveness analysis, implemented.

The intermediate AST can be modified using methods in the SageInterface and SageBuilder libray, which have methods to create new AST nodes and insert, remove or replace nodes in the AST. The transformed code can then be generated by unparsing the modified AST with methods available in the ROSE backend. The modified intermediate AST can also be compiled to machine code using an LLVM extension, but it is not a feature of the main framework.

Chapter 3

Implementation

3.1 Overview

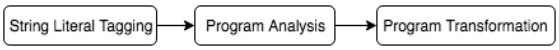


Figure 3.1: Program Transformation Pipeline

Two program transformers have been implemented. The first is a string literal outlining transformer, which gathers all string literals that occur in the program and replace them with global character pointers. It also gets rid of unnecessary pointers to string literals and arrays. The second tranformer takes in a program that has been transformer by the first transformation and shifts string literals to flash memory if they meet certain safety requirements.

Both transformers go through the 3 stage pipeline shown in the figure above. The first stage generates a unique tag for each unique string literal that occurs in the source code. The transformation of the intermediate AST used to generate the transformed source code is done in place using the results of the program analysis.

3.2 Program Analysis

3.2.1 Pointer Alias Analysis

The analysis is written using ROSE's dataflow analysis framework. Within the framework, analysis is done by extending a base *Intraprocedural* dataflow class, then running an interprocedural analysis, which runs the intraprocedural analysis over each function, propagating the states at each call site to the intraprocedural analysis for the function called, and the return states from the intraprocedural analysis back to the caller function's analysis. When a control flow node is first visited, an empty lattice for the analysis is created. The results of the analysis on its predecessor is propagated to it via a meet update. For each function, the analysis is run until there are no more changes to all the lattices in the nodes in the function. The analysis on a finite lattice terminates if the meet update is strictly monotone (Kam & Ullman, 1975).

Pointer Alias Analysis is done via forward propagation, hence the intraprocedural analysis is implemented as a subclass of the IntraFWDataflow class, which traverses the CFG in the forward direction. The analysis is implemented as a modification of the Alias Analysis implementation which available in the projects directory of the ROSE framework repository. The original alias analysis is implemented based on the analysis described in the paper on interprocedural pointer alias analysis by Michael Hind et al (MICHAEL HIND & CHOI, 1999). For each variable, the locations to which they may be aliased to is stored as the alias to the variable.

The analysis is guaranteed to terminate as each time a node is revisited, either aliases are added to its alias set or its alias remains. Since alias set monotonically increases and there are a finite amount of memory locations, the alias sets for all nodes will eventually stop growing and the analysis will terminate.

The original intraprocedure alias analysis had to be modified because aliases to string literals are not captured by the original analysis. In the modified version, placeholders are used for each string literal. In addition, arrays are treated as storage locations as well, so a pointer assigned to an array will have the array as its alias. In the original implementation, arrays assigned to pointers are ignored in setting the pointers' aliases.

The ROSE framework has a Interprocedural dataflow class which runs the intraprocedural analysis over the program in the order of the call graph generated for the program. However, the analysis is flawed as it blindly incorporates the aliases at the return state of the callee functions with that of the caller functions. This is inaccurate in the case when an argument that is not passed by reference gets modified in the callee function. Furthermore, the interprocedural analysis cannot take into consideration the actual order in which functions in the program is run, which leads to inaccurate propagation of aliases for global variables.

The diagram in the following page shows classes in the rose dataflow framework as well as the classes added for the Pointer Alias Analysis, which are coloured in blue. The hollow arrows indicate inheritance, while solid arrows indicate dependency.

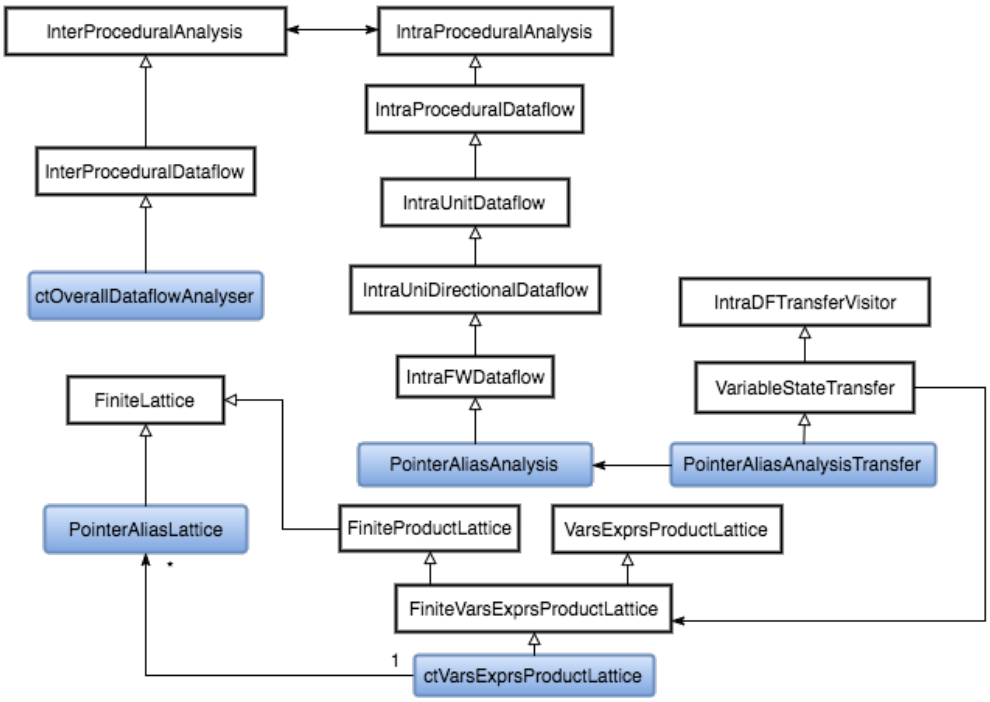


Figure 3.2: Dataflow Framework Classes

For each dataflow node, a *PointerAliasAnalysisTransfer* object is created by the *PointerAliasAnalysis* instance which is running the intraprocedural alias analysis. It visits the AST node corresponding to the dataflow node. It holds a reference to the *ctVarsExprsProductLattice* of the dataflow node and updates it according to the AST node visited.

ctVarsExprsProductLattice stores the result of the alias analysis at each node of the CFG. It is essentially a mapping of variables or expressions in each function to *PointerAliasLattice*s which contain the result of the analysis for each variable or expression. For expressions, the aliases stored in their *PointerAliasLattice*s correspond to the aliases of the variables they evaluate to. For each node in the CFG, its *ctVarsExprsProductLattice* is constructed when it is first visited. Additional

mappings for global variables in the program is also added to the *ctVarsExprsProductLattice*. The initial mappings for global variables are constructed based on their declarations before the interprocedural analysis implemented in *ctOverallDataflowAnalyser* is run.

As both *ctVarsExprsProductLattice* and *PointerAliasLattice* inherit from *FiniteLattice*, both implement the *meetUpdate* method. In the expression `thisLat->meetUpdate(thatLat)`, `thisLat` is updated with the result of the meet of the two lattices. A meet between two *ctVarsExprsProductLattice*s is a meet of their constituent *PointerAliasLattice*s which map to the same variable or expression.

The exact formats of the lattices is in the appendix.

3.3 Transformations

3.3.1 Transformation 1: String Literal Outlining

The transformer first identifies character arrays which have never been modified, based on the results of the pointer alias analyser. These are converted to constant character pointers to string literals. This reduces the runtime SRAM needed in cases where a larger character buffer is created for the unmodified string.

The next step replaces all occurences of string literals with their placeholder variables. In places where a character pointer can be statically resolved to a single string literal, it is also replaced with the placeholder to the string literal. However, the replacement is not done if the pointer's address is accessed at that point. So

in the case of the statement $a = \&b$, even if b is aliased to a single string literal at that point of the program flow, it is **not** replaced with the placeholder for the string literal. The assignments and variable declarations made redundant as a result of the replacements are removed. The definitions for the placeholders are then inserted as global declarations.

An example of the transformation is shown below:

```
const char* testGlobal = "test global string";
void setup() {
  const char * stringsarr[] = {"string1", "string2", "
     last"};
  const char *m = "pointer";
  const char *test1 = "test";

  char arr[] = "str array";
  Serial.println(arr);

  char fArr[10] = "testarr";
  Serial.println(fArr);

  if(random(10) > 5) {
  m = "modified pointer";
  }
  Serial.println(fArr);
```

```
  strcat(fArr, " rest");
  Serial.println(stringsarr[1]);
  Serial.println(arr);
  Serial.println(fArr);
  Serial.println(m);

  m = test1;
  Serial.println(m);
}
void loop() {
  Serial.println(testGlobal);
  Serial.println("Hello");
}
```

Listing 3.1: Original Code

```
const char *f_STRLT_10 = " rest";
const char *f_STRLT_11 = "Hello";
const char *f_STRLT_4 = "last";
const char *f_STRLT_9 = "modified pointer";
const char *f_STRLT_5 = "pointer";
const char *f_STRLT_7 = "str array";
const char *f_STRLT_2 = "string1";
const char *f_STRLT_3 = "string2";
```

```
const char *f_STRLT_6 = "test";
const char *f_STRLT_1 = "test global string";
void setup() {
  const char *stringsarr[] = {f_STRLT_2, f_STRLT_3,
     f_STRLT_4};
  const char *m = f_STRLT_5;
  //Originally Serial.println(arr);
  Serial . println (f_STRLT_7);

  char fArr[10] = "testarr";
  Serial . println (fArr);
  if (random(10) > 5) {
    m = f_STRLT_9;
  }
  Serial . println (fArr);
  strcat(fArr,f_STRLT_10);
  Serial . println (stringsarr[1]);
  //Originally Serial.println(arr); But arr is never
     modified, so it is treated like a const char *
     pointer
  Serial . println (f_STRLT_7);
  Serial . println (fArr);
  //m is not replace since its value here is statically
```

```
    unknown
  Serial . println (m);
  //This is after assignment m = test1. test1 is known
    to be assigned to the string literal ``test''
  Serial . println (f_STRLT_6);
}
void loop() {
  Serial . println (f_STRLT_1);
  Serial . println (f_STRLT_11);
}
```

Listing 3.2: Result of first transformation

3.3.2 Transformation 2: Basic Shifting to PROGMEM

This transformation takes in the program produced by the first transformation. It relies solely on the pointer alias analysis performed on the transformed program, hence not all string literals can be shifted into PROGMEM. In order for all the strings to be safely shifted into PROGMEM, a modified liveness analysis to determine the livetimes of each string literals must be performed. Details of the analysis that must be performed will be discussed in chapter 5.

Only string literals meeting the following conditions are deemed to be PROGMEM safe and shifted:

1. **The string literal is not among the possible return values of any**

function.

In this case, the lifetime of the character buffer the string literal is loaded into must be live beyond the buffer, which means that a global scope character array should be used. However, to efficiently allocate the the global buffer for such string literals, the exact lifetimes of the string literals must be known. Otherwise, the global character buffer would have to be populated at the start with all string literals that require the buffer before the rest of the program is run, which defeats the purpose of shifting the string literal to progmem.

2. **The string literal is not used in any constructor apart from the Arduino String class constructor.**

   This is because of the limitations of the implementation of the current analysis, which does not trace the effects of a class constructor. However, even if the class constructor is analysed, things get dicey if a string gets assigned to an attribute in the function. A character buffer with the same lifetime as the object must then be created. The class definition will have to be modified to have a character buffer the size of the longest string which might be assigned to the attribute as one of its private attributes.

3. **The string literal is not among the aliases of any non placeholder variables.**

   This avoids the scenario where a variable is aliased to multiple string literals, but not all of them have been shifted to PROGMEM.

   With this condition, string literals in arrays of string literals will not be

shifted to PROGMEM as well. It is possible to have an array of strings where some strings are in PROGMEM and others in SRAM. However, the two kinds of strings must be handled differently in the transformed program. In the current implementation, all arrays of string literals are marked as aliased to all the string literals that may be contained in the array. The analysis can be made more precise by attaching an additional datastructure to the PointerAliasLattice for the array which tracks the string literals that may be aliased at each slot. However, there can be instances in the program where the exact array index an operation affects cannot be statically determined; hence the effects of the operation will still be attributed to the whole array. If the granularity of the alias analysis is increased by tracking each slot in the array separately, then the condition for arrays can be relaxed slightly such that in the scenario where is possible to statically determine the slots that will contain only PROGMEM safe strings, the strings which might be aliased to such slots can be shifted to PROGMEM. However, for such arrays which turn out to be mixed arrays, this may result in increased complexity (and code size) of the transformed code as guards will have to be inserted in cases where the exact array index affected by an operation cannot be statically determined.

For non placeholder character pointers, this condition can be relaxed by allocating a character buffer which takes up 1 byte more than the length of the longest PROGMEM safe string the pointer is aliased to. At the point where a string literal in PROGMEM is assigned to the pointer, the string

literal loaded to the character buffer from PROGMEM and the value of the pointer is then set to the address of the start of the string literal in the character buffer.

The usage of the string literals shifted to PROGMEM is then identified. Functions in both the Arduino String library and Print library have been adapted to work with flash strings. Where PROGMEM string literals are used in such function calls in the code, they are cast as $\_\_FlashStringHelper$ pointers and the flash string friendly version of the function will be used for the string literal. Note that because of a missing *StringSumHelper* constructor for type $\_\_FlashStringHelper$, the String + operator does not work if the left hand side argument is a PROGMEM string. The fix for this is to add the line:

```
StringSumHelper(const __FlashStringHelper *str): String(str) {}
```

in the *WString.h* file which will be used in the compilation of the sketch.

An example of the transformation on code that uses only the Arduino Strings library (without character pointers) is shown below:

```
String stringOne, stringTwo;
void setup() {
  Serial.begin(9600);
    while(!Serial){
// wait for serial port to connect. Needed for native
   USB port only
    ;
```

```
  }
  stringOne = String("this");
  stringTwo = String("that");
  Serial.println("\n\nComparing Strings:");
  Serial.println();
}
void loop() {
  if (stringOne == "this") {
    Serial.println("StringOne == \"this\"");
  }
  if (stringOne != stringTwo) {
    Serial.println(stringOne + " =! " + stringTwo);
  }
  stringOne = "This";
  stringTwo = "this";
  if (stringOne != stringTwo) {
    Serial.println(stringOne + " =! " + stringTwo);
  }
}
```

Listing 3.3: Original Code using Strings

```
#define FS(x)(__FlashStringHelper*)(x)
const char f_STRLT_6[] PROGMEM ="This";
```

```
const char f_STRLT_5[] PROGMEM =" =! ";
const char f_STRLT_4[] PROGMEM ="StringOne == \"this\"";
const char f_STRLT_3[] PROGMEM ="\n\nComparing Strings:"
  ;
const char f_STRLT_2[] PROGMEM ="that";
const char f_STRLT_1[] PROGMEM ="this";
class String stringOne;
class String stringTwo;
void setup() {
  Serial . begin (9600);
    while(!Serial){
// wait for serial port to connect. Needed for native
  USB port only
    ;
  }
  stringOne = String(FS(f_STRLT_1));
  stringTwo = String(FS(f_STRLT_2));
  Serial . println (FS(f_STRLT_3));
}
void loop() {
  if ((stringOne == FS(f_STRLT_1))) {
    Serial . println (FS(f_STRLT_4));
  }
```

```
  if ((stringOne != stringTwo)) {
    Serial .  println ((((stringOne)+FS(f_STRLT_5))+
       stringTwo));
  }
  stringOne = FS(f_STRLT_6);
  stringTwo = FS(f_STRLT_1);
  if ((stringOne != stringTwo)) {
    Serial .  println ((((stringOne)+FS(f_STRLT_5))+
       stringTwo));
  }
}
```

Listing 3.4: Result of Transformation

In order for the PROGMEM strings to be used in other function calls, the strings must first be loaded into a character buffer. For each function call, the size of the buffer needed is the size needed to accomodate all the PROGMEM string literal arguments, including the null byte which separates the string literals in the character buffer. The character buffers for the string literals in the function calls are not used beyond the function call. Hence the size of the character buffer needed in each function is the maximum sized needed by the function calls in the function.

Hence, functions with function calls that require string literals to be loaded from PROGMEM are transformed in the following manner:

1. The character buffer is instantiated at the start of the function
2. Before each function call that requires string literals to be loaded PROGMEM, instructions to copy the strings from PROGMEM into the buffer are inserted.
3. The starting address of each string in the character buffer is passed on to the function call.

Here is an example of the described transformation, as generated by the tool.

```
void setup() {
    char const *test1 = "test";
    const char *m1 = testPrint(test1);
    Serial.println(m1);
}
```

Listing 3.5: Original Code using character pointers

```
const char f_STRLT_9[] PROGMEM ="test";
void setup() {
    char f_arrbuf[5];
    strcpy_P(&f_arrbuf[0], f_STRLT_9);
    const char *m1=testPrint(&f_arrbuf[0]/*f_STRLT_9*/);
    Serial.println (m1);
}
```

Listing 3.6: Result of Transformation

3.4 Environment setup

The ROSE compiler must be installed in order to be able to run the transformer. ROSE requires a gcc version of between 4.2.4 and 4.8. In my development, I used a **ROSE 0.9.6a** instance in a Ubuntu 14.04 Docker container setup with the aid of the scripts by Alex Marginean [1]. In addition, the path to the installed ROSE library must be appended to the *LD_LIBRARY_PATH* variable.

Due to a minor bug in the ROSE compiler, function return values for the PointerAliasAnalysis will not propagated if the analysis debug level is set to 0. I've fixed it in my Docker image, which is at `https://hub.docker.com/r/naomilwx/dockerrosebuild`. The source code for the code transformer is at `https://github.com/naomilwx/fyp-arduino-code-transformer`

To run the transformer on Arduino sketches, access to the Arduino standard library is needed. The root path to the library should be updated in the Arduino variable set in the Makefile. The development was done with libraries for the Arduino IDE 1.6.7, and the code generated by the code transformers can be compiled with the **Arduino IDE version 1.5** or later.

3.4.1 Running the Code Transformer

The transformers will be compiled into the ***bin*** folder of main source directory. The first transformer is compiled into an executable called *itransfrom*, the second into an executable called *ptransform*. The second transformer has no effect on code not modified by the first transformer, but the first transformer can be run

[1] https://github.com/AlexMarginean/dockerizedROSE

on its own in this format:

```
bin/itransform -c [includes, each with the -I flag] <file-to-transform>].
```

However, because of the long list of includes needed for the transformer to be able to work with an Arduino sketch, the transformation should be run with the script in *run.sh*. It is run in this manner

```
./run.sh <file-to-transform>  <more-flags-and-non-arduino-lib-includes>
```

The code produced by the final transformation can be found in the ctResult folder in the directory of where the original code is.

3.5 Known Limitations of the Implementation

The transformation can only be performed on a single file at a time. The transformation should be done on the main sketch. The transformation can be performed on other files linked to the main sketch as well, however the resulting transformation might be inaccurate if there are **extern const char *** variables - such variables might get removed prematurely. Also, as the analysis is done after the preprocessing stage, function definitions in source files other than the one being transformed are not available during analysis. The effects of such function can only be approximated - all arrays passed to non constants are assumed to be modified, the aliases of all pointers passed by reference are marked as statically indeterminate.

The source file must be parsable by the EDG frontend. The paths to the files included in the source file must be indicated with the *-I* flag. Paths in to the

standard Arduino libraries have been included in the Makefile, if files from any other library is included, the path to the library, prefixed with the -I flag, must be appended to the arguments to *run.sh*.

Certain preprocessor macros are set by the Arduino IDE during compilation, depending on the board selected. In some cases, not having the preprocessor macros will result in errors during parsing by EDG as certain blocks of code are wrongly ignored. The preprocessor directives are currently set based on the Arduino Uno board as the *ARDUINO_PREPROC* variable. For more accurate analysis when working with code for other boards, the flags should be modified according to the architecture of the board.

The EDG version used by ROSE only supports C++98 and C++11. Hence the transformer cannot work with any code that has features from C++14. The version of the EGD frontend adapted for rose, EDG-4x is not fully C++11 compliant yet. Certain C++11 feature such as override lambdas are not supported.

All macros in the original code are fully subsituted in the transformed code. However, the EDG frontend drops the PROGMEM attribute, hence code already using the F() or PSTR() cast will get mangled.

Chapter 4

Discussion

4.1 Discussion of Transformations

For programs with multiple occurences of the same string, such as format strings, the first transformation alone would result in some savings in memory without additional optimisations by the compiler. However, for programs with no repeated strings the transformation may result in an increase of SRAM usage, especially in cases where the string literals are directly used in arguments to functions, but the transformation creates a placeholder for the string literals, costing an additional 2 bytes (size of character pointer) per placeholder. The additional placeholders may also incur additional flash memory in this instance due to the additional assignment operations.

The second transformation should result in a noticable decrease in SRAM usage in most cases. However there are exceptions to this. An example would be programs that only use string arrays rather than standalone string literals as arrays of string literal are not shifted to flash memory in the current implementation. Exceptions are not limited to programs with string arrays - a constructed exception

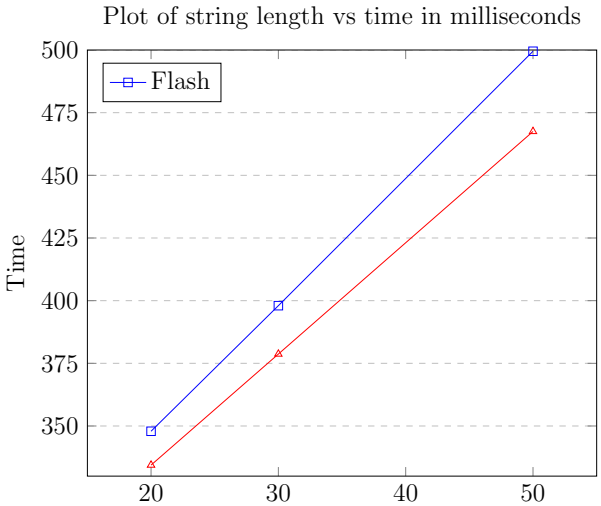
would be one that uses only one or two byte long string literals. However, this is rather contrived and unlikely to happen in practice.

4.2 Potential PROGMEM Time Penalty

Since data must be moved from flash to RAM before they can be used, there might be a noticable time penalty involved in using PROGMEM. To get a sense of the time overhead for AVR microcontrollers, I did an experiment with code of the following structure on the Arduino Uno, recording the time taken when the string to be copied into the array was 20, 30 and 50 characters long.

```
const char testStr[] = "0123456789abcdeabcde";
void runTest() {
    char buff[56] = "stuff";
    strcat(buff, testStr); //Copy string into buffer
}
void loop() {
    //Repeat the experiment 10 times
    for(int j= 0; j < 10; j++){
        start_timer();
        //Record the time taken to copy 10,000 times
        for(int i = 0; i < 10000; i++){
          runTest();
    }
    stop_and_print_timer();
```

```
    }
    while(true) {} //Busy wait to stop the repeated
       execution of loop()
}
```



Length	Flash	SRAM
20	347.9	334.4
30	398	378.7
50	499.5	467.5

Figure 4.1: Average of 10 recorded timings for loading a string 10,000 times

The gradient of each graph gives the amount of time taken to load a character 10,000 times in each case. For the flash string, it takes 5.1ms while for the SRAM string it takes 4.4ms. This translates to a difference of 7 ns per character, or 14μs for 2000 characters, a rather negligable figure for most applications.

4.3 Comparison of performance

To assess the effectiveness of the tool at reducing the amount of SRAM needed by the code, tool was run a number of source files. Both the original source code and the transformed code generated by the tool were then compiled with avr-g++ with optimization set to `-Os`, which is the optimization used by the Arduino IDE for compilation. As the shift to flash memory is only done on string literals, experiments are not run on code without string literals.

4.3.1 Experiments on code from the Arduino examples repository

As source code in the Arduino examples repository exclusively use the String class, all string literals are shifted to PROGMEM by the tool. Hence, the amount of RAM saved is expected to be close to the total size of the string literals in the code. No additional runtime stack memory is needed for character arrays to hold string literals which are arguments to function calls

The test was run on 14 source files. The code were uploaded and run on an Arduino Uno board. Both the compile time estimation of RAM usage (which is the sum of the .data and .bss section sizes) and the runtime measurement of the maximum RAM usage was recorded. The runtime RAM usage was measured using the `MemoryFree` Library[1], which considers both the total amount of memory in the free list maintained by the memory allocator and the amount of space left between the top of the stack and the top of the heap. The table below shows

[1]http://playground.arduino.cc/Code/AvailableMemory

a summary of their combined results. A full breakdown of the results is in the Appendix.

Total Length of String Literals (bytes)	Total RAM Saving (bytes)	Total Runtime RAM Saving (bytes)	Total Additional Flash usage (bytes)	RAM Saving as % of String Literals Length	Ratio of Flash Incurred to RAM saved
2237	2242	2242	3216	95.9	1.43

Table 4.1: Results for Arduino Example Source Code

For some samples, the amount of RAM saved was slightly greater than the total length of the string literals in the original source code. This could be due to padding of string literals for which the sizes are not an integer number of words.

4.3.2 Experiments on code requiring additional libraries

For a more realistic trial, source code with requires external libraries not shipped with the Arduino IDE was obtained from Github. As all the code obtained were written for boards with additional hardware, both the original and transformed source code were compiled but not run on an Arduino board. The source code were obtained from 2 repositories: one is a repo with Flight Telemetry code[2] and the other is a repo with examples for using the Arduino as a switch [3]. The test was done on a total of 7 source code.

[2]https://github.com/landis/arduino

[3]https://github.com/sui77/rc-switch

Total Length of String Literals (bytes)	Total RAM Saving (bytes)	Total Additional Flash usage (bytes)	RAM Saving as % of String Literals Length	Ratio of Flash Incurred to RAM saved
1643	1616	1224	98.4	0.76

Table 4.2: Results for Source Code from Github

In most cases, most of the string literals could be safely shifted to PROGMEM, hence the high ratio of SRAM saved to total length of string literals. For cases where there are very long strings, the amouunt of additional Flash used may be significantly lower than the total length of string literals. This is because the strings themselves do not take up additional flash space. Without the PROGMEM directive, they will be in the .data section instead of the .progmem.data section.

Trying out Mikal Hart's Flash.h library

The WebServer code from the `rc-switch` library was translated by hand to use Mikal's wrappers for placing strings in PROGMEM. The table below shows the comparison of the amount of Flash and RAM used by the original code and the two transformed versions. The total length of String Literals in the original source code is 643 bytes.

Source Code	Flash needed	RAM needed
Original	14,212	1,200
Tool Transformed	14,320	574
Hand Transformed with Flash.h	15,064	704

Table 4.3: Comparison of memory needed in bytes

As expected, the code using `Flash.h` has additional overhead in RAM due to the usage of classes with virtual functions. This example illustrates the potential advantage the tool has over using the abstraction over PROGMEM provided by the `Flash.h` library. The layer of abstraction has overheads in both Flash and RAM.

4.4 Optimisations performed by the compiler

avr-gcc has some optimisations[4] which can be enabled to reduce the amount of SRAM and Flash required to run the code. By default, the Arduino IDE compiles with the flags -ffunction-sections and –gc-sections, which discards unused functions. The compiler also drops unused definitions and data.

avr-gcc is also able to merge occurences of identical string literals and floating point constants. This optimisation is also enabled during compilation by the Arduino IDE. This is similar to some of the optimisations done by the first trans-

[4]https://gcc.gnu.org/onlinedocs/gcc-4.0.3/gcc/Optimize-Options.html

formation. In my experiments, I found that while duplicate occurences of the string literals are indeed merged so multiple occurences of the same string literal do no take up additional SRAM, the same cannot be said for string literals wrapped with the F() macro. So two instances of `F(‘‘helloworld’’)` will take up 20 bytes of memory instead of 10 bytes. Hence the tool provides savings in flash memory over just simply using the F() macro.

Hence, while optimisation at the source code level is less powerful because information from other source files is unavailable prelinking, the current work fills a void where the compiler is unable to optimise. Specifically, the compiler does not shift string literals to PROGMEM to reduce RAM usage, and it does not perform optimisations on string literals in PROGMEM.

Chapter 5

Further work

5.1 Further Transformations

5.1.1 Shifting other unmodified data to flash memory

Shifting unmodified arrays or structures of data to flash memory will also result in significant reductions in SRAM usage, especially if they are global scope arrays. For example, each character pointer in an array of strings takes up 2 bytes (for AVR architectures). So 20 bytes of SRAM will be saved if the array is shifted to flash memory. This can be done regardless of whether the strings themselves are actually in flash memory. The conditions for safely shifting the arrays to flash memory are similar to that of string literals described in section 3.3.2. However, the arrays are passed as function parameters, it may be better not to shift them to flash memory because they would have to be copied to SRAM before the function call. Alternatively, in the case where the function definition is available in the preprocessed source file, a version of the function which operates on flash arrays can be defined. However, this may lead to code bloat exponential to the number of array parameters in the worst case. This is undesirable as it would consume

additional space in flash.

One challenge in shifting constant arrays into flash memory is that the amount of memory required for pointers differ across datatypes and microcontrollers. For example, while a pointer spans 2 bytes in the AVR microcontroller, a pointer spans 4 bytes in the ESP8266(a WiFi module which can be used with an Arduino board), which uses a Tensilica L106[1].

5.1.2 Allocation of character buffer so more strings can be shifted to flash

Two types of buffer (both character arrays) will be used:

1. A global buffer for strings returned by functions or non placeholder global pointers to string literals

2. A local buffer for non placeholder local pointer to string literals

For both the local and global pointers, the amount of memory allocated is 1 byte more than the longest string literal that will be assigned. The memory allocation will be done based on the points in the program flow where the pointers will be used. For functions, the memory allocated will be based on the longest string returned by the function and the allocation will be based on points in the program flow where calls to the function are made.

[1]From the ESP8266EX Datasheet

https://www.adafruit.com/images/product-files/2471/0A-ESP8266__Datasheet__EN_v4.3.pdf

Modified Liveness Analysis

In the standard liveness analysis(Pfenning, 2008), a variable is live if it has been defined and will be used in future. However, in this case the allocation of memory to a pointer should ideally be just before it is first used or just before a branch point in which the pointer will be reassigned (and its reassigned value is used). The analysis can be done by first marking all nodes of interest for each variable - nodes where the variable is used or the dominating successor to a branch point where the variable is reassigned and used. A forward traversal of the CFG is done and a flag is set for the variable at each marked node and all nodes traversed thereafter. A backward traversal of the CFG is then done and the transfer equations at each node n to mark a variable as live is such:

$\text{LiveIn(n)} = \text{Use(n)} \cup (\text{LiveOut(n)} \cap \text{flagged(n)})$

$\text{LiveOut(n)} = \cap\text{LiveIn(successor(n))}$

The analysis will be extended to functions in a similar manner, where traversing the CFG in the forward direction, for each function, nodes starting from its function call are flagged. The transfer equations for the backward propagation of liveness for each node n is the same as before, where each function is considered to be used where there is a function call to it.

Allocation of buffer

For the allocation of the global buffer, the lattices from the liveness analysis will be traversed in forward CFG order, collecting a list of list of functions and global

character pointers which are “live” at the same time. For example, [(a, b), (a, b, f1()), (a, c)].

In the lists, lists whose elements are subset of other lists are filtered. In the example, (a, b) is dropped, producing [(a, b, f1()), (a, c)]. Each list serves as a constrain where the space taken up by their elements in the character buffer cannot overlap. The target is to find an allocation where the maximum space needed for the buffer is minimised.

To reduce fragmentation in the buffer, the following heuristic will be used: longest livetimes first (approximated by the number of lists the elements appear in), followed by longest length in event of a tie in livetimes.

5.1.3 Replacement of Arduino Strings with character arrays

Usage of the Arduino String objects, may lead to fragmentation in the SRAM due to its usage of the heap. For Strings where their maximum possible length can be statically determined (for example in cases where Strings are never extended in a loop where the number of iterations is statically unknown), their usage can be replaced with character arrays.

However, this transformation can be tricky as it must be ensured that all function calls which used the replaced String have a equivalent form that works with character arrays. In some cases, the equivalent function may have to be generated by the transformer, replacing calls to Arduino String library functions with the character array equivalent. For example, the = operator is replaced with

strcmp.

5.1.4 Create a shared buffer for strings that are modified in-place during runtime

Currently, no optimisation is performed if on character arrays that may be modified at any point in the program flow. However, if two local character arrays are not live at the same time, they can use the same memory. So instead of having two character arrays on the function stack, a single array the length of the longer array can be used in this instance.

5.1.5 Merge substrings in PROGMEM

If a string literal in PROGMEM is a substring of another string literal also in PROGMEM, instead of creating a new PROGMEM character array for the string literal, its occurance can be merged with that of the longer string. So portions of the longer string in PROGMEM is read into SRAM whenever the substring is used.

5.2 Other Improvements

5.2.1 Simple GUI and better generalisation of preprocessor flags for other boards

A simple GUI can be built, which allows users to select

i The source file to translate

ii The path to the libraries the source code is dependent for

iii The Arduino board or microcontroller type

For iii, a mapping of Arduino board to preprocessor flags to set should be used but it may be good to allow the preprocessor flags to be configurable by the user as well.

5.2.2 Integration with the Arduino IDE

This can be implemented as a dropdown in the tools menu runs the tool on the active Arduino sketch based on the settings and library paths available to the IDE. However, the challenge in implementing this is the dependency the tool has on the ROSE compiler. The tool requires ROSE to be installed to run, however ROSE currently cannot operate with GCC versions greater than 4.8.4, and ROSE support for windows is currently incomplete.

Chapter 6

Conclusion

A tool has been developed to reduce the amount of RAM required for the execution of an Arduino program. This is an optimisation which the avr-gcc compiler does not perform. As can be seen from the tests run, shifting just String Literals to flash memory can result in significant savings in programs where string literals are heavily used. More savings can be realised by shifting other constant data to flash memory. However, the current implementation is merely proof of concept. In order for the tool to be usable by the casual Arduino user, more should be done to make the tool more user friendly.

References

Barrag'an, H. (2004). Wiring: Prototyping physical interaction design.

Cooprider, N., & Regehr, J. (2007). *Offline compression for on-chip ram* (Technical report). School of Computing, University of Utah.

Daniel J. Quinlan, Markus Schordan, B. P., & Kowarschik, M. (2003). The specification of source-to-source transformations for the compile-time optimization of parallel object-oriented scientific applications. *Springer-Verlag Berlin Heidelberg* (pp. 383–394), , 2003.

Kam, J. B., & Ullman, J. D. (1975). Monotone data flow analysis frameworks. *Acta Infomatica 7* (pp. 305 – 317), , 1975: Springer.

MICHAEL HIND, MICHAEL BURKE, P. C., & CHOI, J.-D. (1999). *Interprocedural pointer alias analysis* (Technical report). IBM Thomas J. Watson Research Center.

Pfenning, F. (2008). Lecture notes on liveness analysis.

Quinlan, D., & Liao, C. (2011). *The rose source-to-source compiler infrastructure* (Technical report). Lawrence Livermore National Laboratory.

Xuejun Yang, Nathan Cooprider, J. R. (2009). *Eliminating the call stack to save ram* (Technical report). University of Utah.

Appendix A

Structure of the Analysis Lattices

ctVarsExprsProductLattice				
Variable/ Expression	PointerAliasLattice			
	Aliased Vars	Alias Relations	Current Alias Status	Variable Status
type: varID Object containing variable/ expression info & type	type: varID set Set of varIDs corresponding to placeholders to storage locations the variable is aliased to	type: Pair (LHS info, RHS info) Used to generate the set of aliases	type: boolean true - known false - unknown Indicates if the variable's alias can be determined at the current node of the program flow	type: enum Bottom Initialized Reassigned Unknown Modified

Table A.1: Dataflow node Lattice for Pointer Alias Analysis

The variable status flag marks variables which have been reassigned at any point in the program flow. For array variables, the flag Modified indicates that the array has been modified at some point in the program flow.

The alias relations field stores alias relations as they are set by the assignment statement. Each member of the pair is in the form (varID, symbol, dereference level). For example, the assignment $a = *b$ will result in the pair: [(varID(a), 'a, 0) , (varID(b), 'b, 1)]. The list of aliases of a will then be updated with the aliases of the aliases of b.

The additional state keeping fields, the *Current Alias Status* and *Variable Status* fields do not violate the property that the *meetUpdate* of a PointerAliasLattice is monotonic. Both fields will be updated to the maximum of the respective values of the two lattices. For the *Current Alias Status* field, false is considered to be a "larger" value.

Appendix B

Detailed results for 4.3.1

File	Strings Size	Flash	SRAM	Max RT RAM	Flash	SRAM	Max RT RAM	RAM Saved	Additional Flash needed	Ratio of RAM saved to String size
Character Analysis.ino	398	3,310	624	636	3,380	210	222	414	70	1.04
StringAddition Operator.ino	102	5,666	334	415	5,940	228	309	106	274	1.04
StringAppend Operator.ino	113	5,256	336	401	5,530	222	287	114	274	1.01
StringCaseChanges.ino	57	4,080	268	332	4,212	210	274	58	132	1.02
StringCharacters.ino	104	4,334	314	419	4,546	210	315	104	212	1
StringComparison Operators.ino	311	6,142	484	1516	6,946	222	270	262	804	0.84
StringConstructors.ino	62	6,818	274	360	7,012	210	296	64	194	1.03
StringIndexOf.ino	529	6024	678	763	6196	210	295	468	172	0.88
StringLength.ino	98	4,294	314	330	4374	218	287	96	80	0.98
StringLengthTrim.ino	109	4272	320	356	4,394	210	240	110	122	1.01
StringReplace.ino	106	5,210	322	437	5422	210	325	112	212	1.06
StringStarts WithEndsWith.ino	221	4,914	418	483	5018	210	275	208	104	0.94
StringSubstring.ino	94	4,328	302	354	4,706	210	268	92	378	0.98
StringToInt.ino	33	4,366	250	266	4,554	216	232	34	188	1.03
Total	2337							2242	3,216	0.96

Table B.1: Detailed results for tests on Arduino example source code

File	Strings Size	Flash	SRAM	Flash	SRAM	RAM Saved	Additional Flash needed	Ratio of RAM saved to String size
Flight Telemetry1	464	41,750	2,243	42,262	1,793	450	512	0.97
Flight Telemetry2	266	8,454	1,250	8,640	964	286	186	1.08
Webserver	643	14,212	1,200	14,320	574	626	108	0.97
SendDemo	92	4,532	489	4,696	401	88	164	0.96
ReceiveDemo_simp	42	3,620	420	3,692	374	46	72	1.10
ReceiveDemo_adv	116	4,216	612	4,290	504	108	74	0.93
TypeA_WithDIPSwitches	20	2,960	208	3,068	196	12	108	0.6
Total	1643					1616	1,224	0.98

Table B.2: Detailed results for tests on code from Github libraries