2011

Simplifying Embedded System Development Through Whole-Program Compilers

William Patrick Mccartney
Cleveland State University

SIMPLIFYING EMBEDDED SYSTEM DEVELOPMENT THROUGH WHOLE-PROGRAM COMPILERS

WILLIAM PATRICK MCCARTNEY

Bachelor of Computer Engineering

Cleveland State University

May, 2005

Masters of Science in Electrical Engineering

Cleveland State University

May, 2006

submitted in partial fulfillment of the requirements for the degree

DOCTOR OF ENGINEERING

at

CLEVELAND STATE UNIVERSITY

May 2011

This dissertation has been approved for the

Department of **ELECTRICAL AND COMPUTER ENGINEERING**

and the College of Graduate Studies by

_____

Thesis Committee Chairperson, Dr. Nigamanth Sridhar

_____

Department/Date

_____

Dr. Yongjian Fu

_____

Department/Date

_____

Dr. Dan Simon

_____

Department/Date

_____

Dr. Wenbing Zhao

_____

Department/Date

_____

Dr. Janche Sang

_____

Department/Date

To my wife Esther...

ACKNOWLEDGMENTS

I would like to thank all those who gave me the opportunity to complete this work. Without the support of many people, this work would not have been possible.

I would like to first thank my advisor, Professor Nigamanth Sridhar, for all of his support. Not only only in an moral and academic advice but also financial support.

My wife, Esther, for her constant and endless support. She has gone beyond putting up with me, and instead encouraged me to work and go to school for the last nine years.

To my young children, Ashley, Billy and Maggie, who have been a continued source of joy and encouragement. They ask wonderfully blunt questions like "Why are you still in school?" and "Why does it take so long for you to write a report?" which force me to understand the importance (or lack thereof) of my work in my life.

To my parents, who not only provided support and encouragement over the years, but also they read and reviewed this work.

To Hassan Varghai and Hosen Varghai who have been an ever flexible employer the past nearly eight years. When I talk to them about school instead of asking "When do you need off?", they ask "When can you work?". Their continued encouragement and financial support have enabled me to go to school and still have a family life.

To my committee members who have always been available not only when asked to be on my committee, but also when they were my professors. Each of their classes has encouraged my research in different ways, and I am grateful for their support.

To my family, friends and classmates who have provided me with a constant stream of support. Many have read and reviewed my work without understanding the technical content, but have provided detailed reviews nonetheless. Many attended my dissertation defense, providing much needed moral support. Without their support, this work would never have been completed.

To the Department of Electrical and Computer Engineering at Cleveland State University, thanks for being my home for the last nine years. Through all three of my degrees, it has always been welcoming and will forever hold a place in my heart.

It is said that it takes a village to raise a child; I pose that it takes a village to raise a doctorate. I cannot image a way I could have completed this work without the support of so many people.

To everyone that I mentioned above and those that I did not:

I am eternally grateful.

SIMPLIFYING EMBEDDED SYSTEM DEVELOPMENT THROUGH WHOLE-PROGRAM COMPILERS

WILLIAM PATRICK MCCARTNEY

ABSTRACT

As embedded systems embrace ever more complicated microcontrollers, they present both new capability and new complexity. To simplify their development, some lessons of computer application development will translate with additional work. This thesis offers one such translation. It shows how whole-program compilers – those that broadly analyze a program's entire source code – can achieve performance gains and remove faults in embedded system applications. In so doing, this yields a novel stackless threading system named `UnStacked C`. `UnStacked C` enables cooperative multithreading without the risk of stack overflows in embedded system applications. We also propose a novel preemption system called Lazy Preemption. `UnStacked C` with Lazy Preemption enables stackless preemptive multithreading in embedded systems. These remove the possibility of thread stack overflows, but also significantly reduces the memory required for multithreading in embedded systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

CHAPTER I

INTRODUCTION

Computer programming has rapidly evolved over the years. Software development techniques have changed the way computer programs are written. The tools and platforms for computer programming have also changed.

Operating systems (OS) have benefited from closer integrations with hardware platforms, and natural evolutions on their own behalf. Inter-process communication, network stacks, Bluetooth stacks, multithreading, memory isolation and much more, have greatly improved operating systems as target platforms for application developers.

Compilers and Integrated Development Environments (IDE) have changed the way people write software. It is not unusual for programmers to make typos and have the IDE or compiler catch the problem before it creates a failure. As well, the ability to rapidly test an application without the risk of damaging the computer is assumed. The ability to test out a failing application without risking either the OS or the other applications running on the same computer, rapidly reduces application development time.

It is easy to argue that computer programming is easier now than it has ever

been before. All engineering students have at least some exposure to computer programming. With high level languages, integrated development environments and widely available libraries, developers have capabilities that were only imagined in the last decade. Productivity of programmers has been improved, with time being spent in actual problem solving as opposed to fixing unintended errors.

1.1 The Problem

Embedded system[1] software developers have not reaped the same benefits from the revolution of computer programming. Integrated development environments for embedded software development are woefully behind their PC counterparts. PC developers are safely removed from implementation details (e.g. stack depth) but embedded system developers enjoy no such safety. While some PC development techniques have transferred, many more have struggled to find a place in the resource-restricted embedded realm. Many of the libraries and OS capabilities simply have not been able to translate into embedded development. Some examples of this are memory separation and virtual memory, which have led to development of libraries and systems which depend on these functionalities. This makes embedded system development lacking compared to their computer programmer counterparts.

Most embedded system software is written using event-driven[2] programming. This allows developers to utilize much of the system resources while avoiding overheads which most computer programmers can conveniently ignore. This often precludes such niceties as multithreading, complex libraries, and, as a result, the ability to reuse code. These limitations are often the artifacts of resource restrictions, which leads to tailoring applications to specific hardware.

[1]For the purpose of this dissertation, we define embedded systems as computer systems with a fixed application and often real-time requirements. We specifically exclude systems which are designed to run multiple applications, such as smart phones.

[2]Event-driven applications are those which are run specifically off of events and event handlers. The term is synonymous with interrupt-driven.

Some embedded systems are simple enough for developers to simply write a single loop which polls as needed. These types of applications are becoming less and less common as requirements of embedded systems are continually increasing.

Microcontrollers come in many different configurations, nearly all of which include support for multiple input/output interfaces (I/O). Most applications require a microcontroller to be interacting with multiple interfaces simultaneously (i.e. USB, RS-232, Ethernet, etc.), while still performing its main task. This leads to many different custom applications with little code re-use. This is exacerbated by inconsistent programming interfaces to platform-specific libraries. It forces developers to craft custom applications for each platform.

The body of research on embedded development is dwarfed by that on computer software development. This is also mirrored in industry. This contrast permits embedded development to profit from crossover techniques. Not only for the embedded system developers of today, but also for current computer programmers in becoming the embedded system developers of tomorrow.

PC application development always has an operating system with memory protection and fault detection included. This allows faults, such as stack overflows, to be caught at run time — and users are then notified. Not only that, but also detailed logs are often maintained by the operating system of some application faults. In an embedded system, storage may not be available to log these faults, making them more difficult to diagnose. In an embedded system, there may be no operating system (or a lightweight operating system without such capabilities) to perform the logging of these faults. In PC application development, these faults are often updated through software patches or updates. In many embedded systems, field updates or patches are difficult, expensive or impossible. So in an embedded system, the cost of a fault on all levels — detecting, diagnosing and repairing — is extremely expensive.

This extreme cost of faults in embedded software leads to a need of a different

set of tools. These tools must detect more faults prior to running the software. As the demand on embedded systems has been increasing with the number of interfaces it supports, it is more important now than ever for such tools. Since hardware platforms are very diverse in embedded systems, a single hardware solution is not viable – a software approach is required. Currently, the research into these types of software tools for embedded system software development have been lacking.

Specifically, current multithreading for embedded systems is implemented in a similar fashion to a PCs multithreading. This ignores the missing hardware components PCs have, which can support fault detection and the software to support logging said faults. The fault detection schemes which do exist cannot detect most faults, and the faults that can be detected only occur at run-time. If a fault occurs in an embedded system at run-time, it is often too late. Instead, some tools or frameworks are needed to help prevent faults at compile time.

1.2 The Thesis

This dissertation defends the following thesis:

1. A flexible whole-program compiler framework can enable building compilers that can detect and prevent certain faults, or perform optimizations, in embedded system software at compile time.

2. Cooperative multithreads can be translated by a compiler into event-driven state machines. These state machines could have lower memory requirements known at compile time, preventing thread stack overflows.

3. Preemptive multithreads can be translated into similar event-driven state machines with the same bounding of memory requirements by using Lazy Preemption techniques.

1.3 The Solution Approach

Our approach starts with a new framework for manipulating programs enabling whole-program transformations named C-XML-C. We have used C-XML-C to implement a series of compiler tools or transforms. Several transforms have been developed to detect faults, prevent faults and optimize performance.

C-XML-C is evaluated by examples. First we evaluate it by writing several example transformations in a variety of different programming languages to show its flexibility and ease-of-use. The next method of evaluation is by implementing transforms for the latter part of this work, and evaluating them together on existing real-world embedded applications. The final portion of the test is evaluating the ability of C-XML-C to compile different types of C source code.

One of these whole-program transforms, named `UnStacked C`, performs a novel transformation from cooperative multithreads into event-driven state machines. We first evaluate its limits on an example application measuring performance, scalability and size. The second evaluation is performed on a set of existing real-world embedded system applications to evaluate the differences in program and memory usage.

`UnStacked C` translates cooperative multithreads into event-driven state machines. These event-driven state machines do not require separate stacks, instead they have memory requirements known at compile time.

This whole-program transformation on cooperative threads is then extended to support preemptive threads. We propose a method to translate preemptive threads into cooperative threads called Lazy Preemption and add support for it into `UnStacked C`. We then evaluate the overhead and effectiveness of this preemption mechanism by measuring its performance and overhead on an example. We then evaluate its program and memory usage on a series of existing real-world applications.

At the time of publication, all source materials, examples and source code will

be made publicly available.[3] This is an effort to try to promote usage and scrutiny of the work.

1.4 Contributions

This dissertation makes several contributions:

- C-XML-C: A flexible source-to-source C compiler framework which is language and platform independent. It also includes a series of example compilers to remove faults and optimize embedded system software.
- `UnStacked C`: A novel compiler based stackless multithreading system which bounds and reduces the memory overhead required for multithreads. It removes certain faults in embedded system software.
- `UnStacked C with LP`[4]: An extension of `UnStacked C` which allows preemptive multithreads to be translated into event-driven state machines with the same memory bounding and memory usage reduction. This translation also removes many possible faults.

1.5 Organization of the Dissertation

The dissertation is organized in the following way: Chapter 2 presents a general background to the problems with preemptive multithreads and stack consumption in embedded systems. Chapter 3 reviews related research and places our work in context. Chapter 4 contains an in-depth look at the architecture of C-XML-C. Chapter 5 explains the methodology of a stackless threading system (`UnStacked C`).

[3] The source code for C-XML-C and `UnStacked C with LP` will be available at `http://www.CXMLC.com` and `http://www.UnStackedC.com` upon publication

[4] There is a patent on memory mapped lazy preemption that has to deal with allowing threads to delay preemption with a flag. This is not related to this work. Since there is no reference to the term except for the title of this patent [30] we have co-opted the term lazy preemption.

Chapter 6 extends the stackless threading system to support preemptive multithreading(`UnStacked C with LP`). The dissertation is concluded in Chapter 7.

If a reader is well-versed in preemptive multithreading and stack usage in embedded systems, than Chapter 2 can be skipped. Readers interested in the details of C-XML-C can focus on Chapter 4. Readers interested in stackless multithreading can focus on Chapters 5 and 6.

CHAPTER II

Background

To understand the details of this research, a firm understanding of embedded software systems is required. In this chapter we will discuss interrupts (Section 2.1), stack usage (Section 2.2), multithreading (Section 2.3), compilers (Section 2.4) and faults (Sections 2.1 and 2.3). Each of these subjects could require books of knowledge to fully understand them; We only cover an overview of the foundations of each. We will also cover some details of the hardware architecture that influences this research.

2.1 Interrupts and Faults

Interrupts are an important part of many systems. They allow a software system to respond to true hardware events in a timely fashion. When an interrupt occurs, it forces the software to branch to a specific location and then execute there. This location is known as an *interrupt vector*. In some systems, the interrupt vector is always a constant value [4], in other systems it is an index to a table that can be changed or even repositioned at runtime [5]. The function or subroutine that resides at the interrupt vector is called an *interrupt handler*. After an interrupt handler is

done executing, it returns, allowing the system to continue executing.

In an embedded system, interrupts are the underlying framework used to implement event-driven state machines. These event-driven state machines can be used to implement concurrent applications or they can be used to implement blocking functions for multithreaded applications. So whether embedded developers are writing event-driven programs or multithreaded programs they must deal with interrupts.

Interrupt usage is plagued with many possible pitfalls. The heart of all of these issues is the sharing of processor resources. This stems from the fact that all interrupts must share the resources of the processor with the rest of the application. This includes not only the program space, RAM and CPU time, but also the registers, the stack, and all peripherals. The sharing of some of these resources: program space, also known as read-only memory (ROM); program memory, also known as random access memory (RAM); central processor unit (CPU) time; and general purpose registers[1] (GPR) can occur naturally. The CPU is shared naturally since only one instruction can be executed at once; if an interrupt occurs, it will begin executing the interrupt as it would the next instruction, returning to that instruction when the interrupt completes. Registers are shared not only between interrupts and the main application, but also between different functions. They are shared by each function storing the registers they are going to use, and then restoring the values back into the registers before they exit. ROM is shared naturally. Since only one instruction can be executed at once, only one location needs to be accessed simultaneously. ROM is then shared by allocating different functions or subroutines in different locations. Most RAM is shared in a similar fashion; different segments of the program use different portions of RAM. If the same segment of RAM is shared between interrupts and the main application, the sharing may not occur naturally. Most of the sharing comes at a

[1]Registers are very small, very fast temporary storage devices processors use to operate directly on. General purpose registers are registers which can be used for any purpose. Much of a given processor's work is manipulating and performing calculations using the general purpose registers.

price either in resource consumption or requiring certain precautions to be taken to avoid different fault scenarios.

A fault is an instance when a system does not execute according to the specifications. In this research we are simply treating the source code as the specification, and identifying a fault as a situation where the application may not follow the source code. For example: $i = i + 1$ should have the effect of incrementing i by one.[2] If the line $i = i + 1$ ever does not increment i then a fault has occurred.

These faults can be significantly difficult to detect and track down using testing techniques. Since interrupts can be generated by external stimuli, they force the execution of the rest of the program to change in new, different and unexpected ways. For instance, fault A only happens if an interrupt occurs exactly during a specific instruction. If the instruction is only executed 1% of the time and the interrupt only occurs once an hour, the fault may never occur during testing. Due to Bernoulli's Theorem (also known as the Law of Large Numbers) if thousands of units are run for 24 hours a day, faults *will* occur in the field at a rough rate of 24% once per day [42]. Prior to discussing all the possible faults, we will first delve into how resource sharing occurs.

When an interrupt occurs, the context[3] of the current running application must be stored prior to the interrupt being executed and then restored after the interrupt is completed. Typically, the program counter of the current location in the application is pushed onto the stack. Sometimes, one or more registers may be pushed onto the stack by hardware also, but these are the exception. Most registers must be pushed onto the stack in the software of the interrupt handler. There is a common optimization: only the registers used by the interrupt handler must be stored and restored. For general purpose registers, C compilers tend to handle this

[2]Ignoring the case of overflows. Overflows are usually well understood, and explained in entry level programming classes. While they can cause faults in some systems, the properties of overflowing are exploited in other systems. We treat overflows not as a fault, but as an exploited feature of the language. In essence, the line $i = i + 1$ really means that it will increment, possibly rolling over.

nicely, pushing the registers prior to using them and then popping them back off after execution is completed. This can be seen in Listing 1.

Listing 1 includes an excerpt from a disassembly of an embedded application. It includes the preamble and postamble of a single interrupt handler named `__vector_16`. In the first part, it pushes registers $r0$ and $r1$ onto the stack. Then it uses *in* to load the status register (number 63 or 0x3f in hexadecimal) into $r0$ and it pushes that onto the stack as well. Then it runs the body of the interrupt handler. After the body it restores (from the stack) the status register, $r0$ and $r1$ in the reverse order. As the interrupt returns, it first restores all of the values of all registers that it used, in this case it is $r0$, $r1$ and the status register.

```
000002fc <__vector_16>:
 2fc:   1f 92           push    r1
 2fe:   0f 92           push    r0
 300:   0f b6           in      r0, 0x3f        ; 63
 302:   0f 92           push    r0
  ..............
 314:   0f 90           pop     r0
 316:   0f be           out     0x3f, r0        ; 63
 318:   0f 90           pop     r0
 31a:   1f 90           pop     r1
 31c:   18 95           reti
```

Listing 1: Excerpt of an interrupt handler from the disassembly of Blink (a TinyOS example program) when compiled for an ATMel ATMega128.

When writing interrupt handlers in assembly language, much care must be taken to ensure that not only are the general purpose registers which are used saved, but also any registers that may change because of side effects. One common example is a status register that includes some flags. These flags may change from instructions which do not reference this register (for instance, an addition may change a carry flag). Except in the case of very simple interrupt handlers, it is common, when interrupt handlers are implemented in assembly language, to see all of the general purpose registers stored in memory at the beginning of an interrupt handler, and then all of

[3]We define an application or a thread's context as all of the local variables, states, registers and temporary values. These are the minimum required to resume program execution where it used to be. Fundamentally, a context describes where we are at, what we are doing, and what we are doing it to.

them restored at the end of the interrupt handler.

In addition to the general purpose registers, peripherals may be shared between the application and the interrupts. This sharing generally requires storing a different set of registers, and more often it requires disabling interrupts during critical phases. The peripherals are not necessarily external devices, it can also be completely internal portions, such as the hardware multiplier for the Texas Instruments MSP430 [69]. This multiplier is treated as a peripheral and much care must be taken to store the correct registers in the interrupt handler. So peripherals that can be used simultaneously in the interrupt handlers and main application need to store the registers with the contexts, but also need to disable interrupts during critical sections.

Disabling interrupts is the technique of turning off, or simply disallowing interrupts to occur. It is done using hardware on the processor which delays the interrupts until they are enabled once again. As shown in Listing 2, disabling and re-enabling interrupts can be done in assembly. It would seem that disabling interrupts is a panacea. If done correctly, briefly disabling interrupts can protect against many types of faults.

```
1 Disable_Interrupts:
2   CLI;
3
4 Enable_Interrupts:
5   SEI;
```

Listing 2: ATMega Assembly code for enabling and disabling interrupts.

Typically, interrupts must be disabled and then restored as opposed to disabled and enabled. This is shown in Listing 3. Disabling (lines 1-6) involves checking to see if interrupts are already disabled (line 3) and storing that result (line 5) prior to disabling interrupts (line 4). When restoring interrupts (lines 7-9), only enable interrupts (line 8) if they were already enabled in the previous disabling. This return value of `disable_int` is saved by the calling function, and then that value is passed to the restore function. This allows interrupt disabling to nest, without creating

a problem with interrupts being accidentally enabled when they are required to be disabled.

```
1 int disable_int(){
2  int status;
3  status = are_interrupts_enabled();
4  DINT(); //Disables interrupts in hardware
5  return status;
6 }
7 void restore(int status){//Takes an argument of previous interrupt status
8  if(status)EINT();
9 }
```

Listing 3: Pseudocode for disabling and restoring interrupts

Keeping interrupts enabled is very important for application accuracy. In many interrupt handlers timers may be read, and if an interrupt is delayed for too long the timer value will have changed. Another possible problem is that interrupts can queue up so that they do not execute in the order they occurred. There is a priority that is preassigned to all interrupts so that when they do execute, they execute in order. Typically, there is a vector number for each interrupt and that number is the priority (lower value is a higher priority) [4]. So when they execute at the same time, the highest priority executes first, then the next highest, and so on. This means that if an interrupt with a lower priority occurs first and then a second interrupt with higher priority occurs while interrupts are disabled, the interrupts will be executed in a different order then the causal events occurred. It is also possible that the same interrupt could occur twice while it is being disabled, so that the first (or the second depending on the hardware) interrupt is essentially lost.

Many processors contain configurable interrupt priorities to override the natural vector priority so that some of the interrupt priority problems can be fixed. This priority control does not fix the temporal problems that disabling interrupts can cause. So it is important to correctly disable/restore interrupts in some sections of an embedded application to prevent faults, but that must be balanced with the need for timely and accurate interrupts to prevent other faults.

If interrupts are not disabled when memory or variables are shared between

interrupts and the main application, two possible data faults[4] can occur. The first fault occurs when a variable needs multiple words to represent its value. This is called a *multi-word data fault*. This can either be a single number, or a structure of multiple values. The second type of data fault is a *temporal data fault*, which loses the computation of either the main application or the interrupt handler because the accesses occur at the same time.

2.1.1 Multi-word Data Faults

A multi-word data fault is shown in psuedocode in Listing 4. This psuedocode assumes that the increment operation can be performed directly on memory (removing other possible faults) and that all of these instructions are *atomic*. An atomic instruction is one that cannot be split in the middle into multiple sub operations. This means that an interrupt cannot occur in the middle of an operation, only before or after an operation. The multi-word data fault can happen when an interrupt occurs after the first byte is incremented and a roll-over occurs, but before the second byte is incremented.

```
1  //Allocate count as 2 words (a 16 bit number)
2  WORD count 2
3  sub inc(){
4   Increment count[0]
5   //If there is a roll-over, then data faults can occur until line 7
6   if(roll-over){
7     Increment count[1]
8   }
9   return
10 }
11 sub main(){
12 start:
13  call inc()
14  goto start;
15 }
16 sub interrupt(){
17  //save context
18  if(count <= 10){
19    //Error, do something special
20  }
21  //restore context
22  return_from_int
23 }
```

[4]What we call data faults are sometimes refered to as *data race conditions* [80]

Listing 4: Psuedocode for multi-word access data fault.

Listing 5 shows the trace of how the program is expected to run, but Listing 6 shows how a data fault can occur. The trace shown in Listing 5 successfully rolls over the count, from 255 to 256, without causing any faults, because an interrupt did not occur during the at-risk section (when time is eight and nine). It is easy to identify the time of the fault by looking at the values of *count* [0] and *count* [1] over time. In Listing 6 we see the cumulative value of count go from 254, to 255, to 0 and finally to 256.[5] The trace in Listing 6 differs starting at time 9 when the interrupt occurs.

```
1  time instruction           count[0] value      count[1] value
2  1    goto start            count[0]=254        count[1]=0
3  2    call inc()            count[0]=254        count[1]=0
4  3    Increment count[0]    count[0]=255        count[1]=0
5  4    test roll-over        count[0]=255        count[1]=0  No Rollover in last add
6  5    return                count[0]=255        count[1]=0
7  6    goto start            count[0]=255        count[1]=0
8  7    call inc()            count[0]=255        count[1]=0
9  8    Increment count[0]    count[0]=0          count[1]=0
10 9    test roll-over        count[0]=0          count[1]=0  Rollover in last add
11 10    Increment count[1]   count[0]=0          count[1]=1
12 11    return               count[0]=0          count[1]=1
```

Listing 5: Execution Trace without a fault code. Software operating correctly.

```
1  time instruction           count[0] value      count[1] value
2  1    goto start            count[0]=254        count[1]=0
3  2    call inc()            count[0]=254        count[1]=0
4  3    Increment count[0]    count[0]=255        count[1]=0
5  4    test roll-over        count[0]=255        count[1]=0    No Rollover in last add
6  5    return                count[0]=255        count[1]=0
7  6    goto start            count[0]=255        count[1]=0
8  7    call inc()            count[0]=255        count[1]=0
9  8    Increment count[0]    count[0]=0          count[1]=0
10 9    Interrupt occurs
11 10    test count <= 10     count[0]=0          count[1]=0    //Test count is 0, so
12 Fault occurs, count should either be 255 or 256
13 11    return_from_int      count[0]=0          count[1]=0
14 12    test roll-over       count[0]=1          count[1]=0    No Rollover in last add
15 13    return               count[0]=1          count[1]=0
```

Listing 6: Execution Trace with a multi-word data fault code. Software operates incorrectly.

This problem can be extremely difficult to track down, since it will only occur when an interrupt occurs in a two instruction window. Since it occurs so rarely, it may

[5]The effective value of the two byte number *count* is calculated by multiplying $256 * count\,[1]$ and adding that to *count* [0]. In the case of *count* $[0] = 0$ and *count* $[1] = 1$ the cumulative value of count is 256.

not be caught in testing. The real issue is that to the interrupt, the value appears to be correct, while really it is not. To prevent these faults in the source code, disabling and restoring of interrupts must be added only around the critical region. A *critical region* is a portion of the application where a fault may occur. The changes required to prevent the data fault are shown in Listing 7.

```
1  //Allocate count as 2 words
2  WORD count 2
3  WORD istatus 1
4  sub inc(){
5   istatus = interrupt_status()
6   DISABLE_INT
7   //Start of critical section
8   Increment count[0]
9   //If there is a roll-over, then data faults can occur here until after the second
        increment
10  if(roll-over){
11    Increment count[1]
12  }
13  //End of critical section
14  RESTORE_INT(istatus)
15  return
16 }
17 sub main(){
18 start:
19  call inc()
20  goto start;
21 }
22 sub interrupt(){
23  //save context
24  if(count <= 10){
25    //Error, do something special
26  }
27  //restore context
28  return_from_int
29 }
```

Listing 7: Psuedocode for multi-word access data fault: fixed with a critical section

2.1.2 Temporal Data Faults

The pseudo instructions from the previous examples oversimplified the problem. Most modern microcontrollers use *load-store* architectures, which typically require separate instructions to read the data from memory (loading) then modify it with other instructions. They finally store back the registers to memory [58]. These architectures do not have the same atomicity in their operations on memory that our previous ex-

amples did.[6] Since load-store architectures require multiple instructions to accomplish the same thing, these operations are no longer atomic. These systems are susceptible to another data fault known as a temporal data fault.

These temporal data faults occur when a processor loads a variable into its registers, modifies it and prior to writing it back, an interrupt occurs, which overwrites the previous value. After the interrupt completes, when the application then writes the value, it will simply overwrite any changes that the interrupt handler made. An example program to demonstrate these faults is shown in Listing 8.

```
1  WORD count 1
2  Main(){
3  start:
4   load count, r1    //Load count into register 1
5   increment r1        //increment register 1
6   store r1, count    //Write register 1 to count
7   goto start
8  }
9  Interrupt(){
10  //Store context
11  load count, r1    //Load count into register 1
12  if(r1 > 200){
13    load #0, r1        //Load a 0 constant into register 1
14    store r1, count    //Write register 1 to count
15  }
16  //restore context
17 }
```

Listing 8: Pseudo code for load-store temporal data fault

The fault occurs if an interrupt happens between the load and the store instructions in main. It is only a fault if count is greater than 200. For instance, if count is 250 and an interrupt occurs between the load and store instructions in main, then the interrupt will write `count=0` out to memory and then main will write `count=251`. If the interrupt was supposed to reset the count (for instance when a physical event happened) then it may not do so. The fixed source code that protects the critical section is shown below in Listing 9.

```
1  WORD count 1
2  Main(){
3  start:
4   istatus = interrupt_status()
```

[6]No architectures have perfect atomicity in all statements in C, so temporal data faults can take place in any processor architectures.

```
5  DISABLE_INT
6  //Start of critical section
7  load count, r1    //Load count into register 1
8  increment r1        //increment register 1
9  store r1, count    //Write register 1 to count
10 //End of critical section
11 RESTORE_INT(istatus)
12 goto start
13 }
14 Interrupt(){
15 //Store context
16 load count, r1    //Load count into register 1
17 if(r1 > 200){
18   load #0, r1        //Load a 0 constant into register 1
19   store r1, count    //Write register 1 to count
20 }
21 //restore context
22 }
```

Listing 9: Pseudo code for load-store temporal data fault - with protection

Temporal data faults can occur far more frequently in C since the compiler has some latitude to implement optimizations [39]. One of the most common optimizations occurs when a compiler loads a variable at the beginning of the function, but does not write the value until the function exits. For instance, the code shown in Listing 10 may never get past the loop. This is because the compiler can choose to load `test_var` when `main` starts, and then never read it from memory again. In this case, it is not an issue with the width of the data, since it is irrelevant if the access to the variable is atomic or not.

```
1  int test_var=0;
2  int main(){
3   while(!test_var){
4     //Loop here
5   }
6   //Start application
7  }
8  void interrupt(){
9   test_var = 1;
10 }
```

Listing 10: C code for possible fault

To understand the specifics of this fault, we need to understand how the compiler may choose to implement the source code. Listing 11 shows one possible optimization choice a compiler could conceivably choose to implement. In this case, it chose to convert line 3 of Listing 10 into lines 3 and 4 of Listing 11. This optimization occurs often, a compiler chooses a variable and stores its value in a register. Since

registers are faster to read, it will execute the loop significantly faster than if it had to read from memory every iteration. Since the interrupt only writes the variable `test_var` and `main` only reads the register `r0`, the loop will never exit.

```
1  int test_var=0;
2  int main(){
3   register r0 = test_var;
4   while(!r0){
5     //Loop here
6   }
7   //Start application
8  }
9  void interrupt(){
10  test_var = 1;
11 }
```

Listing 11: Possible optimized C code which can create the fault

To fix the fault, two main changes should be made. The compiler must be notified that the value of `test_var` can be changed at any time.[7] The compiler is notified that `test_var` can be changed at any time by the keyword `volatile`. The `volatile` keyword essentially turns off certain optimizations which could create the temporal data faults. The updated code is shown in Listing 12 with interrupts disabled.

```
1  volatile int test_var=0;
2  int main(){
3   int status;
4   status = disable_int();
5   while(!test_var){
6     restore_int(status);
7     status = disable_int();
8   }
9   restore_int(status);
10  //Start application
11 }
12 void interrupt(){
13  test_var = 1;
14 }
```

Listing 12: C code fixed from data fault

It is important to only mark variables as `volatile` if it is actually required, since many optimizations will be removed around any computations where this vari-

[7] Since `test_var` is only being used as a flag, it is possible that interrupts do not need to be disabled during accesses, since there is no chance of data fault (if the interrupt executes during a read, the value is simply a flag so even a partial write could count as a complete write). This is a nuanced point and for the sake of correctness versus speed it should be ignored and protected accordingly.

able is involved. If this is abused, it can significantly slow down performance along with a significant increase in the compiled code size. A simple example of source code is shown in Listing 13. If the variables `x`, `y`, `z` and `total` are marked as `volatile` it takes longer and it is larger[8] as shown in Table I.

```
1  #define MAX 10
2  volatile int x,y,z;
3  volatile double total;
4  int main(){
5    asm("nop");
6    for(x=0;x<MAX;x++)
7      for(y=0;y<MAX;y++)
8        for(z=0;z<MAX;z++)
9          total += 1;
10   asm("nop");
11 }
```

Listing 13: Simple C benchmark to measure the impact of volatile

	Code Space	Execution Time
Without Volatile Variables	76	222.9 ms
With Volatile Variables	83	230.34 ms
Percent Increase	9.21%	3.34%

Table I: Simple C benchmark results for the impact of volatile

The results of Table I show that marking variables as `volatile` have a distinct impact on program execution time and code space. Each loop iteration must load the counter out of memory prior to incrementing its counter and then storing the value after the fact. These operations occur in each iteration and take time to execute. These operations also take up code space. These overheads mean that care should be exercised in deciding which variables to mark as `volatile`.

These data faults have quite a bit in common. They both require accessing the same memory locations in both interrupts and the main application. So simple rules can be put together to detect them. Since assembly code is not portable, we will only focus on detecting these faults in C. Since C does not have normal load and store operations, when we talk about a write, we mean a C statement where the variable

[8]The program was compiled for the ATMega88 using WinAVR version 20080610. The exact execution time was measured using AVR Simulator @ 4MHz inside of AVR Studio version 4.14 build 589.

in question is on the left-hand side. When we talk about a read, we are referring to a C statement where the variable is on the right-hand side. These simple rules are shown below [29]:

1. If an interrupt writes a variable, then all accesses (reads and writes) in the main application must be protected.

2. If an interrupt only reads a variable and the main application writes it, then all writes to this variable in the main application must be protected.

3. If a variable is written in neither the application nor the main loop, then it can be read in both safely (i.e. constants).

4. If a variable is only accessed in the main application or only in the interrupts, but not both, then all accesses are safe (assuming only one interrupt can execute at a time).

To protect variables against these data faults, the shared variables must be marked as `volatile`, as well as disabling interrupts during the protected accesses. On certain platforms some accesses are atomic (as in faults that can never occur on certain platforms), but this is the exception. It is better to write portable code in the general case instead of marginally faster, platform specific code. For instance, it is a common misconception that an "int" or a "word" in C is a physical word[9] in the processor. This is not always the case. For instance, in AVR-GCC and AVR-LIBC, which is the compiler and standard library respectively for AVR 8-bit microcontrollers, it defaults to 16-bit ints and 16-bit words [77]. This means that "word" level access in C is not necessarily atomic. No operation in C can be assumed as atomic, unless written in assembly.[10]Even in cases where the physical word is the same size as the C word, accesses are not necessarily atomic, and cannot be relied upon to protect

[9]A physical word is the width of a given register on the processor. It is important to know, because that is usually the limit of operations that can possibly be atomic for a given processor.

against faults. Worse yet, relying on a compiler to generate atomic accesses means that the compiled results may change with compiler optimization level and even compiler version.

It is better to have safe and portable applications than to have one optimized specifically for one single platform, so we discourage any developer from going outside of these rules. One counter-example to these rules are flag variables that are set once in the interrupt handler: these may not cause a fault in certain cases. If the protections around this variable are in place and no fault is possible, then around this one variable some optimizations may not take place, but the program will still run predictably (if not at maximum speed). If the protections around this variable are not in place and a fault is possible then speed is achieved at the cost of correctness. These rules may over-generalize the problems, but since undetected faults carry a high risk and are often too nuanced for developers to reliably detect, we believe it is better to simply focus on checking along these rules.

2.1.3 Interrupt Complexities

In our previous discussions, we have been referring to interrupts with the idea that only one interrupt can occur at any time, and that it will not be interrupted by any other interrupts. In reality there are several different conditions where interrupts can fire inside of other interrupts. The simplest case of this is when one interrupt is executing and another interrupt executes. This is called *interrupt nesting*. If interrupts can preempt (or interrupt) each other, than another rule is required:

5. If interrupts can nest and an interrupt writes a variable and that variable is accessed in another interrupt, then all accesses to this variable must be protected.

[10]Using assembly language does not make data accesses atomic, but when using assembly the data accesses are all explicitly made. This means that developers are aware of what is a single instruction and therefore atomic.

This rule means that any time one interrupt can preempt another interrupt the reads and writes of any variables shared between them must be protected from each other. This idea can be translated into an even more complicated system of interrupts by looking at the rare case when an interrupt can preempt itself. An interrupt that can self-nest is known as being *reentrant*. Reentrant interrupts are extremely rare in production systems and are usually unintended [62]. An additional rule is needed to support data fault protection in systems with reentrant interrupts. If interrupts can be reentrant and an interrupt writes a variable, then all accesses in that interrupt must be protected. Since reentrant interrupts are usually unintended, we will not include this rule in our list of rules.

Another type of interrupt is one that cannot be disabled, and this is known as a *non-maskable interrupt* (NMI). NMIs are often reset vectors and error vectors. The problem with them is they cannot be disabled so normal precautions cannot apply. If an application must have NMIs, then writing them in assembly langurage with minimal shared variables is strongly suggested. NMIs can be artificially created in software on platforms that have a priority interrupt controller. The application could choose to only disable interrupts below a certain level, leaving higher interrupts enabled so they can act as NMIs. These artificial NMIs suffer the same complexities as traditional NMIs.

2.2 Stack Usage in Embedded Systems

A vast majority of embedded systems require a *stack* allocated in RAM. The stack stores not only the return addresses of the currently executing function, but also any local variables and temporary values which a function requires. On some architectures, it also includes the previous stack frame on it.

Prior to explaining what goes into a stack, we must first understand some

general architecture ideas. A stack pointer points to an address in RAM that is the current working location of the stack. It can grow by either increasing or decreasing the address, depending on hardware architecture. On some platforms stack operations are controlled by the callee[11], on others the stack operations are controlled by the caller.[12] These operations are responsible for cleaning up the stack after a function is called. For the purposes of this discussion, we talk about this as the callee being responsible for cleaning up after itself. One other important hardware idea is that in all architectures there is a *program counter*. A program counter is a register that contains the address of the next instruction. When a function is called, the program counter is stored on the stack so that it can be restored when the function returns.

A *stack frame* is a subset of the stack that contains all of the local variables, temporary values and return values of a given function. To help explain what is in a stack frame, see the source code in Figure 1.

Figure 2 shows the stack frame of the `calculate` routine. The `calculate` routine simply performs some calculations on the three arguments and returns the results. There are four main parts to this stack frame. The first part of the stack frame contains all of the arguments. These arguments are setup by the caller with the proper value. The next part of the stack frame is the return address. The return address is the address of the program counter that the function will branch to when it ends. The stack must also store any registers which will be used by the function. These registers may be used by the caller, so their values should be restored prior to branching to the return address.

The final part of the stack frame are the local variables. Not all local variables are defined by the developer; the compiler has the option to add temporary variables needed for calculations. For example, the compiler may translate the source code for `calculate` into something like Listing 14. This shows how the compiler may

[11] A callee is a function that is called by another function.
[12] A caller is the function who called the other function.

```
1 void calculate(int x, int y, int z){
2  int area;
3  area = x * y * z + y / z;
4  return area;
5 }
```

Figure 1: Source code of a function named `calculate` doing arithmetic.

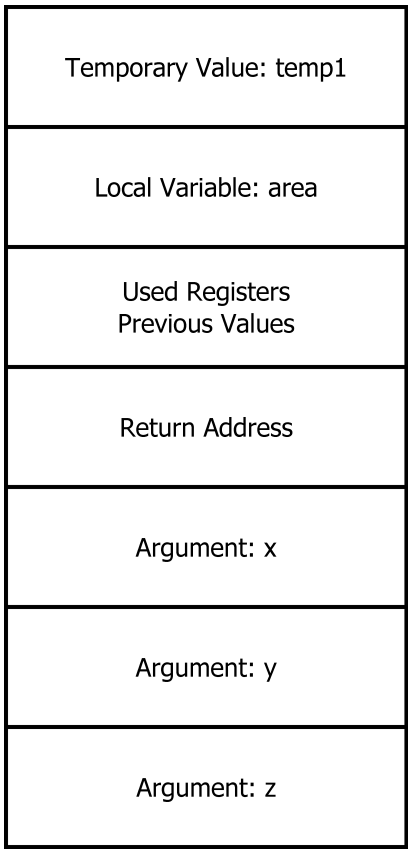


Figure 2: Stack frame of the function `calculate`

need to create a temporary variable, `temp1`, to hold a value. Some processors can perform a single math operation per instruction. That means if a single line of source code requires multiple calculations to implement it, then it will require multiple instructions and possibly temporary values to implement in machine language. Other processors require many instructions to perform a single calculation. For instance, an 8-bit processor performing a 16x16[13] multiplication requires many more instructions than it would for an addition. Between these sets of instructions, temporary values are usually required. Often times registers are used for this purpose, but other times there are more values than the registers can hold.

```
1 void calculate(int x, int y, int z){
2  int area;
```

[13]The notation 16x16 signifies that both of the multiplicands are 16-bit numbers.

```
3  int temp1;
4  temp1 = y / z;
5  area = x * y;
6  area = area * z;
7  area = area + temp1;
8  return area;
9 }
```

Listing 14: A possible implementation of the function `calculate` that a compiler might choose.

The examples from Figures 1, 2 and Listing 14 show possible stack frames, but it is important to realize that the stack frames will change depending upon architecture and compiler options. For example, different processors have different hardware capabilities. This includes not only the size and number of registers, but also the ability of the platform to perform C statements. This stems from the fact that in most architectures, a single line, or statement, in C code takes multiple instructions to execute. The interim values in these instructions cannot always fit inside of the registers, so more temporary variables must be allocated.

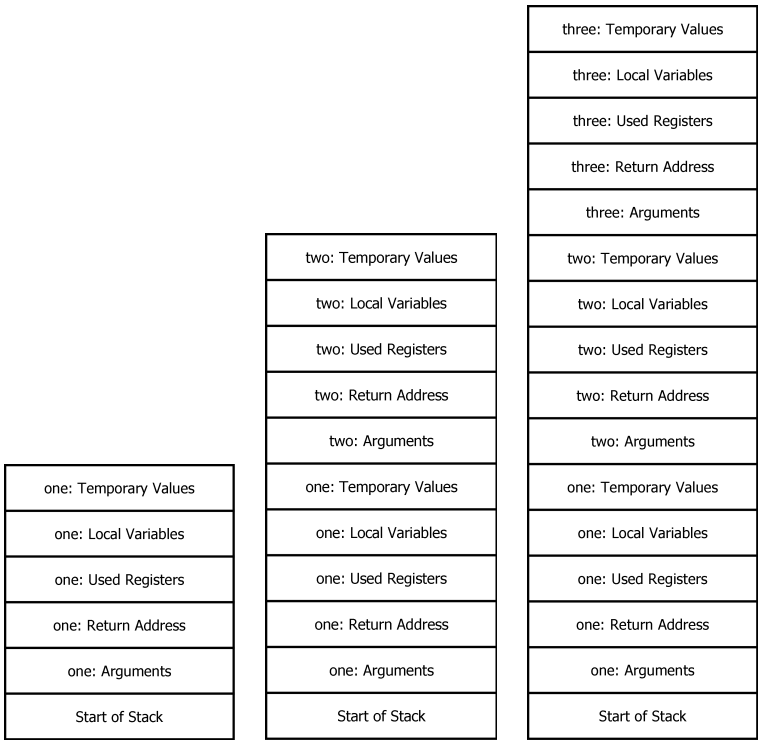
Compilers can change the amount of stack space they use per function. For instance, a compiler may optimize out a temporary variable in one version and not in a different version of the same compiler. Sometimes compilers will use more or less stack space if optimizations are turned on. For instance, a compiler may create a temporary variable to hold some interim calculation which will then speed up the execution of the program, since it does not need to recalculate that interim value multiple times.

```
1  int one(){
2   //Start operation
3   two();
4   //Other operations
5   return result;
6  }
7  void two(){
8   //two.s operations
9   three();
10 }
11 void three(){
12 }
```

Listing 15: Example code of three functions calling each other.

Since each function needs a stack to execute, when they call other functions,

that pushes the stack out deeper. For instance, Listing 15 contains three functions. Function `one` calls function `two` and function `two` calls function `three`. When only function `one` is running, the call stack looks like Figure 3(a). When functions `two` and `three` are running the call stacks can be viewed in Figures 3(b) and 3(c) respectively. Notice when function `two` is executing, the stack frame for function `one` must be stored in a separate location. When function `three` is executing, the stack frames for both functions `one` and `two` must also be stored. This means that the average stack usage is not important. The only stack usage we must be concerned about is the maximum, or worst case, stack usage.



(a) View of stack when program just started.

(b) View of stack after first function invocation.

(c) View of the deepest call depth.

Figure 3: Stack frames of a single program in multiple states.

Compilers can also manipulate the call stack, but it is generally done to improve execution performance. One such technique is called *function inlining*. Function inlining is the act of replacing a function call with the body of the function to be

called. This can improve performance and can remove the need to actually create another stack frame. Depending on the function, it may still require additional stack space (i.e. for temporary values). This stack space is then allocated by manipulating the stack pointer.

Besides calling other functions, functions may also manipulate the stack themselves for a variety of reasons. One reason a function might manipulate the stack is to allocate a user selectable amount of memory. Typically, applications require dynamic amounts of memory for a variety of reasons. Memory is usually allocated outside of the stack using a function called `malloc`[14] and then freed via a function called `free`. Instead of these normal memory allocation techniques, it is possible to allocate memory from the stack by using a function called `alloca`. `Alloca`[15] operates by pushing the stack out further, allowing the user to use this space. The memory is freed automatically when the function returns. In general, calls to `alloca` can be dangerous and should be used with caution. In an embedded system, calls to `alloca` should be avoided since it removes any bounding on the stack size.

Another reason that a function might directly manipulate the stack is because it has a variable number of arguments, or that it is calling a function with a variable number of arguments. The last reason a function may manipulate the stack is done in multithreading libraries. To achieve multithreading, direct manipulations must be done to swap stacks. Swapping stacks is done to switch from executing one thread, which has its own stack, to a different thread, which has a different stack. These manipulations must be done in assembly, since there is no way to swap stacks directly in C.

Compilers generally do not provide any output on the stack frame requirements for each function. Even if they did, they still do not provide a call graph. A call graph

[14] `Malloc` and `free` are part of the standard library that is used to allocate and free dynamic memory outside of the stack [25]. These are a part of the standard library in C.

[15] `Alloca` allocates dynamic memory from the stack [23]. It is freed automatically when the calling function returns.

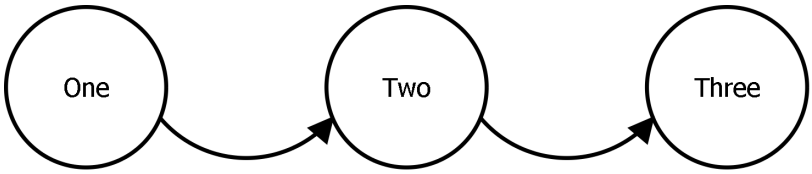


Figure 4: A simple call graph.

is a graph with a node for each function, and a directional edge from each caller to a callee. An example of a call graph that corresponds to Listing 15 can be found in Figure 4. Since neither the stack frame nor the call graph are provided by compilers, the stack requirements for a given program are not known.

Stack usage is complicated by interrupts. Since interrupts can begin executing at any time, they push the stack out further. If the interrupts nest then the stack gets pushed further still. This means that the interrupt call graph and stack frames must also be considered when allocating the stack [63]. Stack requirements are both a function of the interrupts themselves and any functions they call. They also change based upon the enabling or disabling of interrupts throughout a given program [63].

Multithreaded systems have a much larger stack problem than event-driven systems. In event-driven systems, all events share a single stack. In a multithreaded system, each thread must allocate its own stack. This means that the call graphs must be taken into account for each thread individually. To help explain the impact of a separate stack, we will first explain how multithreading is implemented.

Multithreading is implemented, at its core, by using a yield routine. A yield routine is also known as a stack swapping routine. This routine typically pushes all of the registers onto the stack (along with the program counter) and then swaps stacks. This allows one thread to call this yield routine, and have another thread start executing. After swapping stacks, it pops all of the registers and the program counter off of the stack, resuming the context and continuing executing where it was before. This is done through a yield function [50] as shown in psuedocode in Listing 16. Since the routine only does stack operations, it is safe to call from multiple threads

simultaneously. The scheduler only needs to setup the `last_stack` and `new_stack` variables to manage the accesses to the rest of the system.

In a multithreaded system, a scheduler controls the order of execution of threads. A scheduler is a piece of software that decides which thread should run next. It performs this by running immediately prior to any stack swapping operations. This examples is ignoring the details of scheduling because it is not important to this research.

After the `yield` routine runs, the previously running thread has its entire context pushed onto the stack so it contains a full context. Then the top of the stack is stored into the variable `last_stack`. Since the rest of the context is already pushed onto the stack, only the pointer to the top of the stack needs to be stored. After the stacks are swapped, the stack pointer now points to a different context (another thread that called `yield` some time earlier). The registers are then popped off of the stack in the opposite order; restoring the values. The previous values (which were restored) are from a time previously when this thread called `yield`. The program counter is already pushed onto the stack when the `yield` function is called, so it does not need to be pushed on separately. Once the `yield` function returns (popping the program counter from the stack), and the new program counter is in place, the next thread context is fully restored and it continues to execute where it left off.

```
1 yield(){
2  //Here the scheduler runs if needed
3  push General_Purpose_Registers
4  push Status_and_Special_Registers
5  last_stack = stack_pointer_register
6  stack_pointer_register = new_stack
7  pop Status_and_Special_Registers
8  pop General_Purpose_Registers
9 }
```

Listing 16: Pseudocode for Context Switching

Figure 7 contains details of the stack swapping operations between two threads named `foo` and `bar`. These Figures show the processor registers and both stacks as

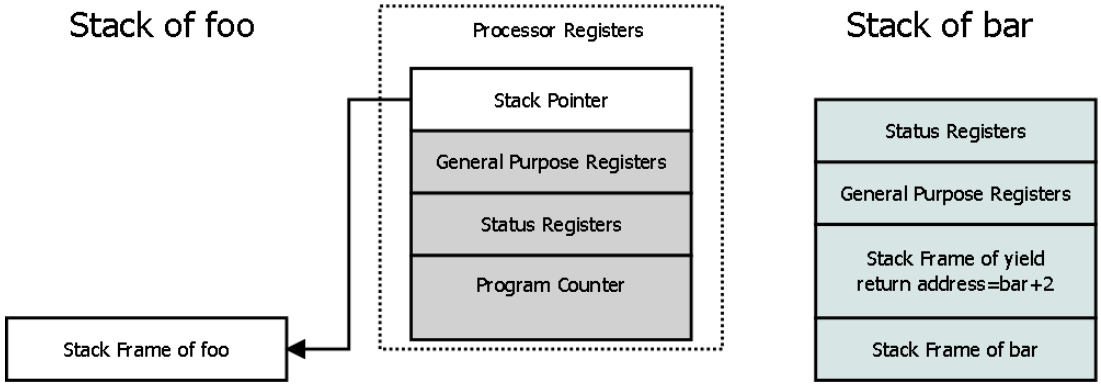
a stack swapping operation occurs between the two threads. Thread `foo` is currently running and thread `bar` is ready to run. Figure 5(a) shows the system prior to running `yield`. Figures 5(b), 5(c), 6(a), 6(b), 6(c) and 7(a) show the swapping operations in detail. Finally, Figure 7(b) shows the thread `bar` executing after the stack swapping is complete.

The routine yield implies *cooperative threading*. To add preemption, a separate interrupt can be set up to force an execution of the yield routine. Interrupts preempt the main application's execution and threads are part of the main application. So interrupts will preempt any threads that do not have interrupts disabled, and if one of the interrupts calls `yield`, it will force a context switch[16] to a different thread.

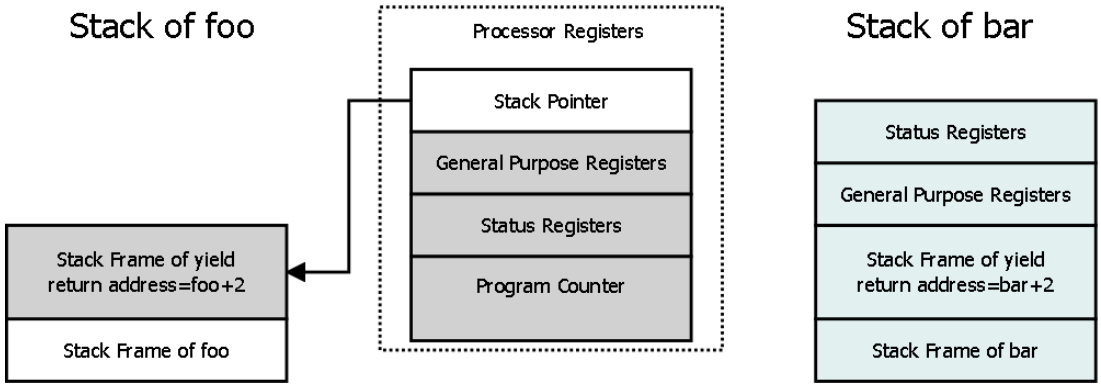
In a single threaded or event-driven system the stack is often allocated so that it will have the remainder of free RAM. If the stack grows too much it causes a *stack overflow*. A stack overflow is when the stack continues to grow to the point where it begins to overwrite other memory. In embedded systems, RAM is typically limited, so stack space is in competition for the limited RAM with the application itself. Stack overflows are actually common in embedded systems. Labrosse said that "whenever someone mentions that their application behaves strangely, insufficient stack size is the first thing that comes to mind" [43]. Lamie also talks about stack overflows: "The results are very unpredictable, but most often include an unnatural change in the program counter" [21].

Since stack overflows can create such havoc in an embedded system there are some things in place on PCs to help prevent them already. The best way to detect and handle a stack overflow is in hardware with a memory management unit (MMU) [58]. An MMU can detect when the stack usage requests a new page, and can therefore know exactly when the stack is running out of space. Since most embedded systems

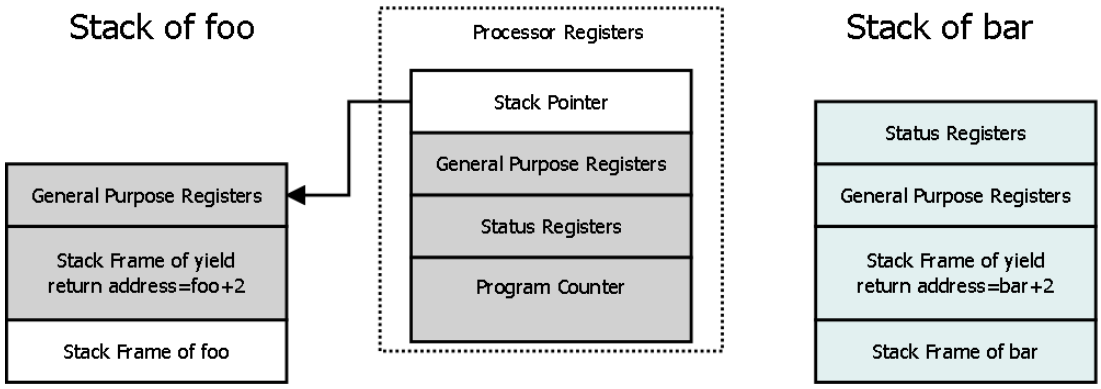
[16]In a multithreaded system, a context switch is exactly the same as a stack swap. A stack, when it is not currently executing, is referred to as a context. This is because all of the current registers and other portions of running logic are pushed onto the stack prior to swapping.



(a) Running `foo` not yet entered `yield`

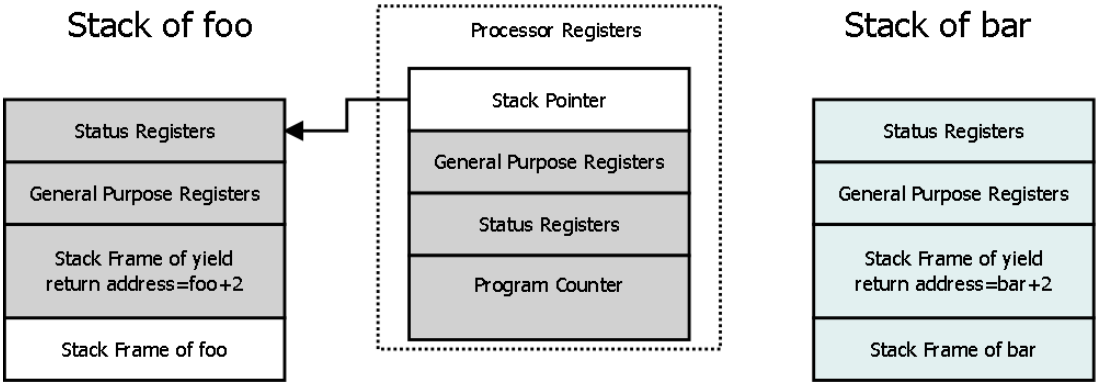


(b) Inside of `foo` entered `yield`

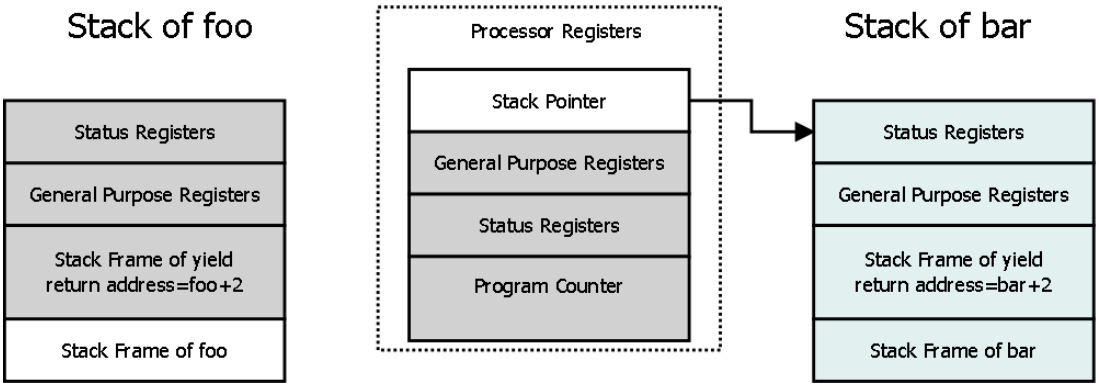


(c) `foo`'s general purpose registers have been pushed onto the stack.

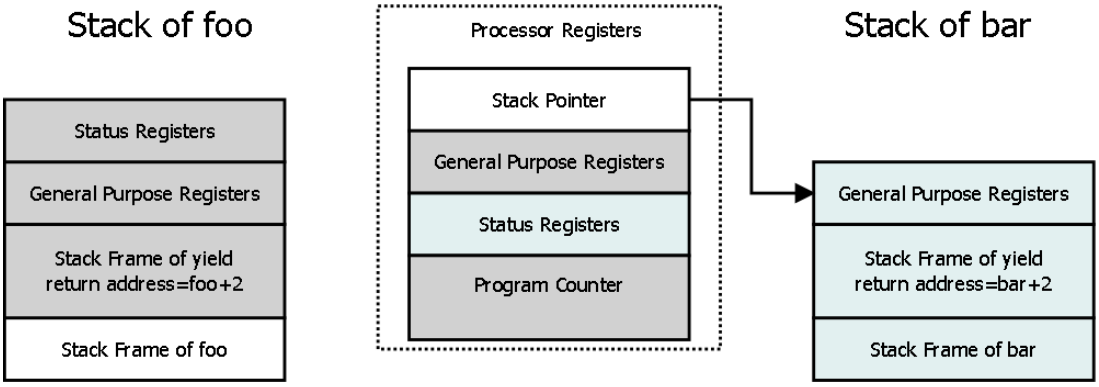
Figure 5: Stack swapping operations of the yield between threads `foo` and `bar`.(page 1 of 3)



(a) `foo`'s status registers have been pushed onto the stack.

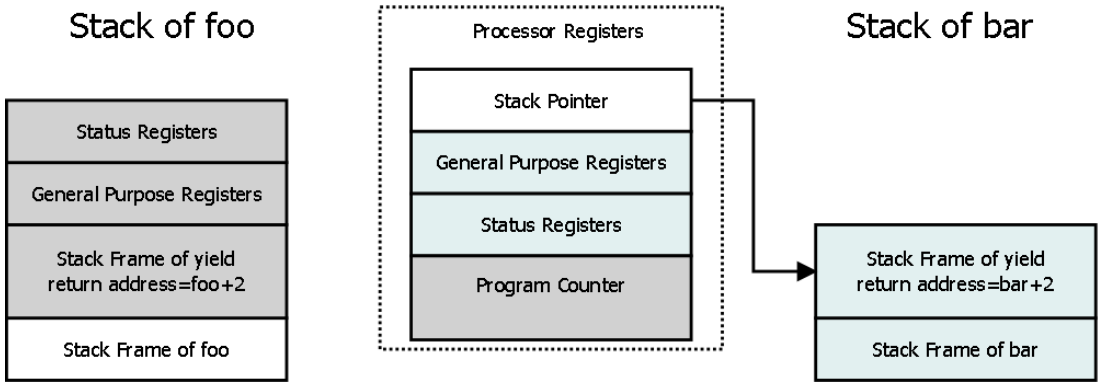


(b) Stack pointer now points to `bar`.

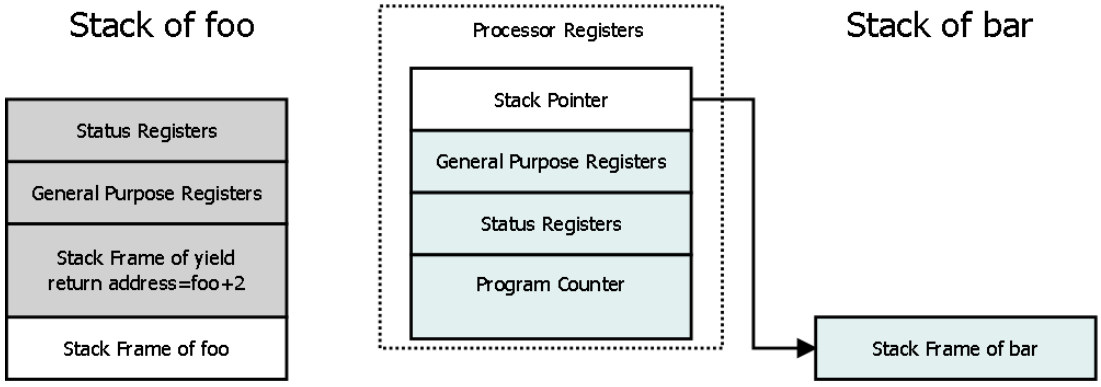


(c) `bar`'s status registers have been popped from the stack.

Figure 6: Stack swapping operations of the yield between threads `foo` and `bar`.(page 2 of 3)



(a) `bar`'s general purpose registers have been popped from the stack.



(b) `yield` has returned into the function named `bar`. Now `bar` can continue executing.

Figure 7: Stack swapping operations of the yield between threads `foo` and `bar`.(page 3 of 3)

do not have an MMU, a different approach must be taken.

2.2.1 Stack Sizing

There are several different approaches for determining the stack size. The first approach is to make an educated guess based upon the source code of the program. The second approach is to measure the current stack size used in a single execution. The third approach is to measure the maximum stack depth used in a single execution. The final approach is to statically analyze the program to determine the maximum theoretical stack consumption of every possible execution.

To make an educated guess a developer should determine the call graph and guess at the results. This means that a developer must have an extremely in-depth understanding of not only how many local variables they used along with the call graph, but also how much stack space the temporary values the compiler will use. If any libraries are used, their stack requirements should be understood too. This is not practically possible, but Labrosse notes that a margin of 1.5 - 2x any estimates should be used to ensure stack safety [43]. In more general terms, Ganssle describes "the standard, scientific way to compute the proper size for a stack: Pick a size at random and hope" [41].

This issue with stack usage is that interrupts create non-determinism in the execution of the program. Interrupts are not guaranteed to execute at specific times; They execute based upon external stimuli which leads to non-repeating program executions. This non-deterministic execution leads to a non-deterministic stack consumption.

An approach better than an educated guess is to measure the actual stack usage at a given time. There are two methods to reading out the stack usage. The first method includes instrumenting the source code with checks to read out the stack pointer at known times (i.e. at interrupt handlers or on context changes) [64]. The second method is by running a debugger and setting break points throughout the source code and inspecting the stack pointer when the debugger hits those break

points. The problem with either of these approaches is that we are only looking at the stack usage at specific times, not at the maximum stack usage of this run.

If a debugger is already going to be used, then getting the maximum stack usage of a single execution is not too difficult. First initialize the stack to some known value. Then run the program for some extended amount of time (or through some set of inputs, etc.) and then look at the stack space itself. One can inspect which values of the stack have not changed, and that is the stack space which went unused [22, 41, 43, 64]. Some integrated development environments (IDE) take advantage of this technique and automate this method for the developer [55]. This method still cannot take into account all of the possible execution paths of an embedded system. For instance, an error condition may be extremely rare (and not come up in testing), but will require 100 bytes more stack space than the maximum seen in testing. This would lead to a stack overflow only during the rare conditions, which could occur in a production system.

The final approach to sizing stacks is to use software to statically analyze the program to determine the required stack space. This approach was developed by Regehr et al. [63]. In our previous work on building TinyThread, we used their technique to calculate the required stack size for threads in TinyOS [50]. There are really two separate techniques which can be used. The first technique is to assume that the interrupts cannot become reentrant and that all the interrupts can preempt each other. This means a simple sum of all of the interrupts stack requirements gives the total interrupt stack requirements. This added to the stack requirements of the main application would give the required stack [12, 22]. The second technique requires checking to see when interrupts are enabled and disabled and evaluating the maximum required stack space [63]. This technique gives a smaller, more accurate stack requirement, but it requires specific knowledge of the interrupt preemption setup of a given embedded system to be correct. If these assumptions are not correct,

than the stack sizing will create sizes too small and stacks can overflow.

Statically analyzing a program to get the stack size is more difficult than many tools would leave you to believe. The first issue is recursion. Unless the recursion is bounded, the stack cannot be analyzed. Yang et al. mention that microcontrollers make limited use of recursion, so this is not a problem [78]. Recursion is detected by both of these tools and the user is notified accordingly (this includes indirect recursion) [63, 50]. The second issue is manual stack manipulation by functions. This is usually done through `alloca` or through an array sized by an argument — these would manipulate the stack in a dynamic fashion which cannot be statically analyzed unless looking backwards at all of the routines which called said function. Manual stack manipulation is extremely rare in any systems, and even more rare in embedded systems so this issue is not a problem.

```
1  void A(){
2   B();
3   C(1);
4  }
5  void B(){
6   C(0);
7  }
8  void C(int x){
9   if(x){
10    D();
11  }else{
12    //do nothing
13  }
14 }
15 void D(){
16 }
```

Listing 17: Source code showing how a call graph can lead to stack overestimation.

Static analysis can have limitations, such as naively looking at the worst case condition for every function. Listing 17 shows an example application where the worst case stack frame appears to be:

$$Stackframe_A + Stackframe_B + Stackframe_C + Stackframe_D$$

This can be from the stack analyzer being overzealous because when C is called from

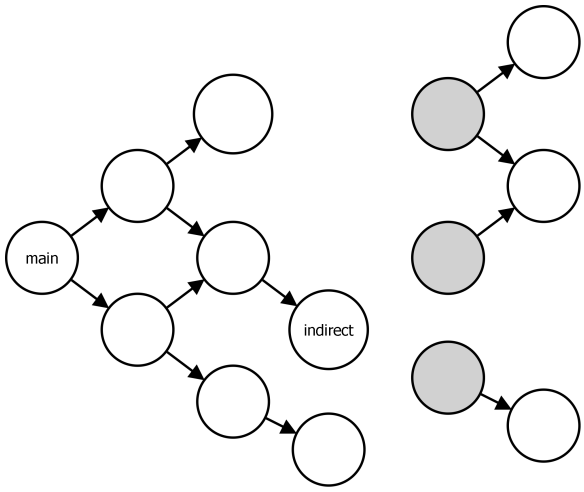


Figure 8: A call graph of an application missing the indirect calls.

B it does not push the stack frame out for D. So the worst case stack frame may actually be:

$$Maximum(Stackframe_A + Stackframe_B + Stackframe_C,$$
$$Stackframe_A + Stackframe_C + Stackframe_D)$$

This means that the worst case stack frame of C does not occur when it is called from inside of B.[17]

Another common problem of static analysis is the use of indirect functions.[18] It is generally possible to instrument these programs to detect the posting of tasks by a real-time operating system (RTOS), but it must be specifically written to support whatever idioms the compiler uses to implement the attaching of tasks to the RTOS [63, 50]. The problem with this approach is that each operating system may have a different method of attaching tasks to their program, this means that the compilers may end up using different idioms.

[17]This conditional stack consumption is very common in nesC code since it generates dispatch tables for event handlers when a parameterized interface is used.

[18]Indirect function calling is when an application stores a functions address in a variable, and then indirectly calls whatever function is stored in that variable.

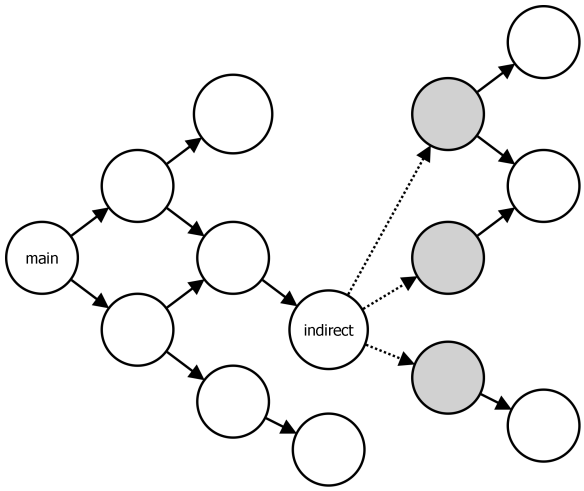


Figure 9: A call graph with links added to the indirect calls by adding links from indirect calls to any uncalled functions.

One solution to the function pointer problem is to make a list of uncalled functions inside of the stack analyzer, and then assume that all of the functions can be called from all of the function pointer locations throughout the program. Figure 8 shows a call graph of a given program that uses indirect calls, but those calls are missing from the call graph. This approach is used to add the missing links as shown in Figure 9. This approach might be slightly overzealous in the sizing of stacks, but it will not under size the stacks with one small exception. If a function is called both by function pointer and directly by the application, then the stack analysis can fail to recognize a possible growth in the stack size. One possible true call graph with all indirect links shown in Figure 10 could possibly lead to errors since the indirect call may have deeper stack requirements then initially assumed.

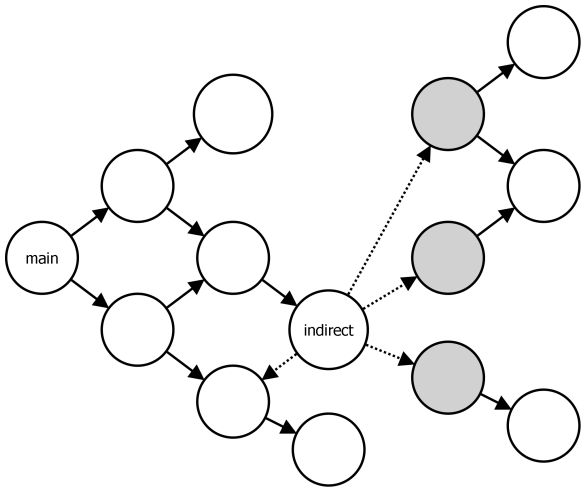


Figure 10: A call graph with all links correctly added. Notice the indirection calls a function which is also called directly elsewhere in the program.

2.3 Multithreading

Multithreading is the idea of running multiple threads of execution at the same time whom all share memory. Multithreading is generally classified as *cooperative* and *preemptive*. Cooperative threads run until the given thread decides to yield, either implicitly because it makes a blocking call,[19] or explicitly through a `yield`. In preemptive multithreading, the above is true and it is extended by an upper bound on time. If a preemptive thread does not yield within some time frame, it is preempted and taken off of the currently executing stack.

Multithreading allows programmers abstractions to share the processor doing seemingly multiple activities concurrently, but it does so with risk. As with interrupts, shared variables between threads can create faults. If a variable is shared between cooperative threads only temporal faults are possible. This risk can be mitigated by marking the variables as `volatile`, which will force writes prior to the processor yielding [39].

[19]Blocking calls are calls to functions which typically starts an operation and puts the thread to sleep until that operation completes.

When using preemptive multithreading, both temporal and multi-word faults can occur. Since preemption is typically not disabled by a thread (thereby defeating the purpose of preemption), protection of shared resources is typically done using a mutex lock of some kind. Mutexes are locks which allow only a single thread to hold at a time. If a second thread attempts to lock a mutex, it will block until the first thread releases it. This allows accesses to a shared resource to be controlled by the mutex.

Even in the face of these complexities in sharing variables and resources, preemptive threading is the standard threading technique in both the PC and embedded multithreading systems. The reason preemptive multithreading is popular on PCs is because a single misbehaving thread is less likely to take down the entire system. Embedded systems follow suit for the same reasons, but we believe this is also because of the momentum of the PC development community. Preemptive multithreading creates the most benefits for a trade off [66]. Since hardware is ever-improving these trade-offs make more sense over time.

2.4 Compilers

Compilers are applications that take in source code and convert it into machine code[20] [6]. Compilers are fairly complicated with many books written on the subject. They must parse the source code files and build up an *abstract syntax tree* (AST).[21] An AST contains the source code elements in a tree data structure so that it is easier for the machine to reason with it. It contains all of the usable information of the source code, but it is missing portions not important to generating machine code (i.e. comments, whitespace, etc.). This AST is then converted into an object code.

If there are multiple files of source code, then each file is compiled individually.

[20] This is specific to C compilers. There are many other types of compilers that output byte code or similar things. Since our focus is on the C language, we will ignore these details.

[21] Abstract syntax trees are also called *Parse Trees* in other literatures [6].

This generates an object file full of object code (often having the ".o" file extension). These objects are then linked by a tool called a *linker* [6]. The linker merges multiple object files and fills out the missing information for any references between the object files. Library files can also be passed to the linker, to include references to any routines used by the other object files. Figure 11 shows a flowchart of a typical compilation process.

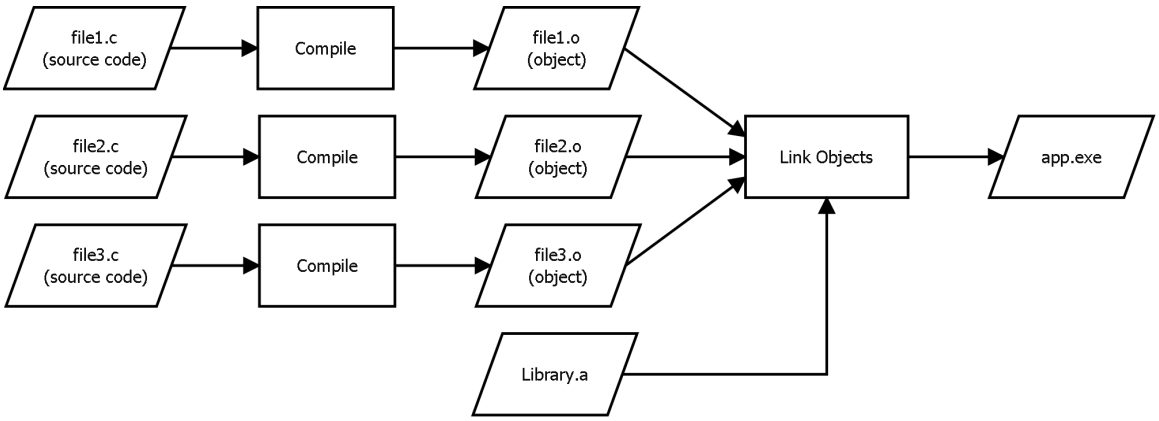


Figure 11: This flowchart shows how files get compiled when using normal compilers.

2.4.1 Whole-Program Compilers

Whole-program compilers are compilers that merge all of the files together in the source code, or the AST stage [54]. This is shown in Figure 12. Whole-program enable different types of analysis and optimizations to be performed on programs. Since the files are all compiled together, there is no way for incremental compilation to be used; If a single file changes, then all files must be recompiled. This is in contrast to the normal compilation process, where only the changed file needs to be recompiled since the rest of the files are in object format already.

Another problem with whole-program compilers is interaction with libraries. To have a true whole-program compilation, the source code of the libraries are re-

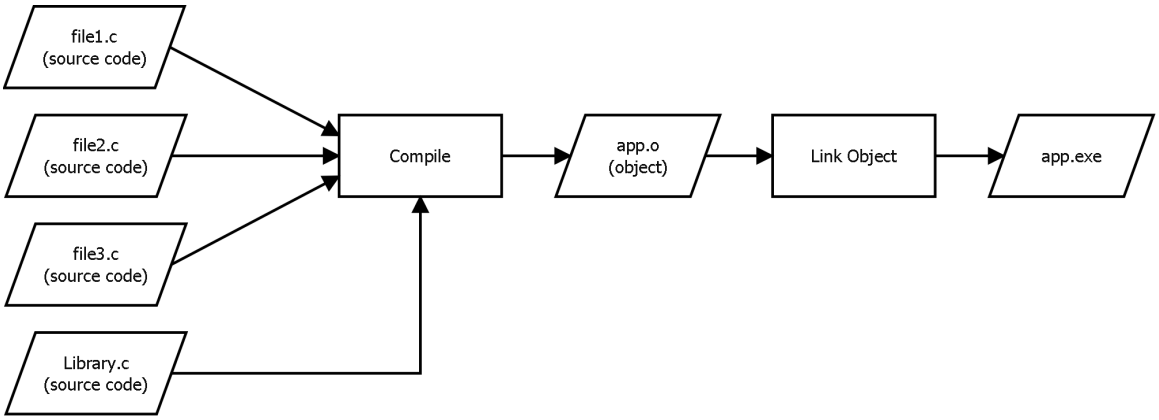


Figure 12: This flowchart shows how files get compiled when using whole-program compilers.

quired. In some cases library source code is not required, but then it is only a whole-application compilation, not a whole-program compilation. In this research we focus on whole-program compilations, although some of the work would translate naturally into whole-application compilations.

2.5 Summary

In this chapter, we reviewed the use of interrupts and their interactions in embedded systems. We reviewed the complexities associated with determining stack size. We also reviewed three types of faults: stack overflows, temporal data faults and multi-word data faults. The latter two can occur when any data is shared between the interrupts and the main applications. They can also occur between any two preemptive threads. We also explained how to protect against these faults using interrupt disabling and mutexes for interrupt and thread sharing respectively.

CHAPTER III

Related Work

The related research can be broken into two major groups — compiler frameworks and multithreading systems. The compilers related work found in Section 3.1 will situate the context for C-XML-C. The context for `UnStacked C` and `UnStacked C with LP` will be setup by Section 3.2. Prior to getting into the multithreading systems, we will first talk about compilers.

3.1 Compilers

There are several different efforts to produce transforming compilers. These can be broken into two main categories. The first category includes source code processors that are used to obtain information or facts from the source code. These are used for analysis, and are not used for code regeneration. The second category includes source code processors that manipulate and transform the program in source code form.

Low Level Virtual Machine (LLVM) falls somewhere in between the two categories [46]. It is used to compile languages into a Low Level Virtual Machine, during which it can manipulate and analyze the program. This can perform some of the

same operations as C-XML-C, but it only outputs LLVM code. There is a C backend, which can output C code instead of LLVM bytecode, but there are many limitations. This does not lead to a good method of manipulating a program in situ. Instead many optimizations are lost that the compiler may have performed, since the source code is first translated into a translation unit prior to modification.

Another example of similar projects that can fall in both categories is the GCC Plugins Project [24]. This is an add-on for the popular GNU Compiler Collection (GCC) that allows developers to add run-time plugins to GCC. This sounds like an ideal scenario, with the exception that it does not document an API, nor does it have any intention to maintain API stability of any kind. In fact, many GCC developers talk about this as a feature to dissuade commercial interests from writing plugins [59].

There are several projects which only analyze the source code. TUanalyzer is a source code analyzer for checking C++ templates [31]. It uses a GCC feature to dump its Translation Unit (TU) which includes much of the Abstract Syntax Tree (AST). It uses the output of this to analyze templated source code.

Another pair of similar projects are CPPX [17] and XOgastan [2]. CPPX translates C/C++ source code into Graphical eXchange Language(GXL). GXL is an XML format for storing graphical information [37]. XOgastan similarly translates the AST into XML. It uses GCC to dump the TU and then converts that into XML. Both of these are useful for extracting information for source code analysis, but neither contains enough information to regenerate the source code.

Another source code to XML translator is the GCC-XML project. It has no built-in support for function bodies, so it only outputs type information and function declarations. In our first attempt at building C-XML-C we added support for function bodies but it still does not have enough information to regenerate the source code.

One source to source translation framework is called Stratego [73]. Stratego is not only a translation framework, but also a language for performing said transla-

tions. It is interesting because it is not limited to the C language. Instead, Stratego transformations can be performed on multiple programming languages. It limits developers because the new transforms must be written in the language Stratego and they must understand Stratego's AST of the application to modify it.

One source to source translating framework that can regenerate the source code is called CIL [54]. CIL allows developers to create AST translations on actual C source code. It is written in OCaml and requires developers to use OCaml to write transformations. It also requires developers to regenerate AST components for any new code they want to inject. So in essence it requires developers to understand the AST, and to be able to generate said AST to be able to translate it.

Meister et al. have extended CIL to support the generation of the AST in eXternal Data Representation (XDR) and eXtensible Markup Language (XML) [51]. Their work is comparing the processing of source code translated into XDR and XML formats. They show that source code in XDR is roughly 10 times smaller than source code in XML format. They also show how it is significantly faster to parse XDR than it is to parse XML. We based our C-XML-C on their XML generator, but we decided not to use XDR for translating the source code. We used XML since it has ubiquitous support across programming languages and it is human readable. Neither of these are the case for XDR. We sacrifice compile time efficiency for ease of use and flexibility.

Yang et al. have developed a way to remove the call stack and save RAM [78]. Their approach is to inline all of the functions. Normal inlining replaces the call to a function with the entire function body. Instead of inlining the functions they call their operation *lifting*. First they copy the function into the caller. Then every additional call, they use goto statements to enter and return from this function body. Their inliner only copies a given function once, and then uses branch statements to execute it from different locations. The general idea is that all functions can be lifted into the

`main` function, so that the compiler can perform intrafunction optimizations, which it performs better than interfunction optimizations. Then it lifts local variables from the stack into the global memory. It operates on single threaded programs.

Yang et al. implement this in CIL as a source to source translation, similarly to `UnStacked C`. It reduces stack consumption by a single threaded application, where `UnStacked C` reduces stack consumption by threads in a multithreaded application. Besides these similarities the transformation itself is orthogonal to `UnStacked C`, since it only transforms a single thread down to a smaller stack size. In a multithreaded system it is functionally an aggressive inliner, and assumes that each function is only executed from the main routine — so it would fail to inline anything that is used from multiple threads.

3.2 Multithreading

A number of articles have been published in the area of concurrent programming [26, 74, 75, 27, 10], and in particular, focused on the debate between event-driven and multithreaded programming [76, 1, 45]. In this section, we will review some of the salient pieces of work as they relate to the ideas we present in this dissertation.

Adya et al. [1] discuss the essential differences between event-driven programming and multithreaded programming. They clarify the distinction in terms of how tasks are managed and how the stack is managed. Event-driven programming involves cooperative task management and manual stack management, while typical (preemptive) multithreaded programming involves preemptive task management and automatic stack management. They present a system of cooperatively scheduling tasks (*fibers* in Windows), while managing the stack automatically. Adya also defines *stack ripping* the act that developers must go through when writing procedural programs in event-driven system. Instead of using a stack to manage the execution

of these procedural operations, they must manually rip the stack into global state variables.

3.2.1 PC Centric Multithreading

Capriccio [74, 75] is a system for servers that completely eschews the event-driven paradigm, and instead adopts the position that support for multithreaded programming can be more efficient. To efficiently manage the space allocated for thread stacks, this system uses a special method called *linked stacks*. The *linked stacks* are based on the observation that in the common case, most threads only use a small portion of the stacks allocated at any given time. The stacks are managed such that they can dynamically grow and shrink depending on runtime needs of individual threads. Capriccio is scalable to 100,000 threads in an application.

The *staged event-driven architecture (SEDA)* [76] takes an opposing view in that threads are hidden from applications. Instead, services are decomposed into *stages*, each of which contains a *thread pool*. Stages are non-blocking and are event-driven. Control transfer from one stage to another is managed using a queue, which serves as an execution boundary. The stages are designed to be self-contained modules with little data sharing across stages. This makes reasoning about SEDA behavior simple. Programmers do have to deal with learning to program in the event-driven paradigm.

Tame [45] is a system that enables programmers to write event-based programs in C++ without having to worry about stack ripping. The Tame system provides a set of primitives in libraries that allows programs to be written as though they were using threads. As such, this work is quite similar to ours in that the Tame primitives result in code that looks similar to `UnStacked C` code. The Tame primitives translate what look like blocking method calls to simple event-driven continuations. The big difference between Tame and `UnStacked C` is that our system does not require the

programmer to use new syntax and new idioms: existing multithreaded C code can simply be recompiled into `UnStacked C`.

Stackless Python [71] is a modified version of Python which does not store any state information on the C stack. Stackless Python adds a type of microthread which are usually stackless and scheduled cooperatively in terms of a C stack. Newer versions of stackless python adds some additional functionalities to support preemption and using the C stack only when necessary.

3.2.2 Small Embedded Systems

TinyOS [35] is a popular operating system for networked embedded sytems. Programs for TinyOS are written using *nesC* [29], a dialect of C that provides a component-oriented veneer suited for building sensor network applications. TinyOS is purely event-driven, and explicitly eschews multithreading. All operations that would normally block, such as messaging, sensing, etc., are implemented as *split-phase* operations following a classical event-driven paradigm. As a result, all TinyOS programmers have to deal with stack ripping.

Several systems have been proposed that challenge the TinyOS position on events vs. threads for building embedded system applications. *Protothreads* [20] are a way of programming embedded systems running the Contiki operating system [19] using a limited form of stackless threads. They are limited in that Protothreads do not support automatic local variables, and state shared across multiple threads must be stored globally. In spite of this disadvantage, the big advantage of Protothreads over other threading solutions for embedded systems is its extremely small memory footprint. Protothreads are implemented in headers via the C preprocessor. In its output, `UnStacked C` uses a similar technique as that used by Protothreads.

Our previous work is *TinyThread* [50], a full-functional threading API for TinyOS that enables programmers to write cooperatively-threaded programs. This

library is much more heavyweight than protothreads since each thread requires its own stack. Stacks in TinyThread are automatically managed, which means that programs can use local variables, and high-level synchronization constructs across threads. Although the thread library is accompanied by a tool that provides tight estimates of actual stack usage, the memory requirement places a severe limitation on the number of threads that can be accommodated on typical sensor hardware. We have implemented a modified version of TinyThread that *significantly* reduces the memory overhead while retaining its flexibility found in Chatper 5.

TOSThreads [44] extends *TinyThread* with a new API and preemption. It adds runtime loading of threaded applications, but it breaks TinyOS safety in that TinyOS event-driven commands can be potentially be preempted if they are called from a thread. It uses message passing to provide a layer of isolation between multithreaded code and event-driven code, but this is not enforced by the compiler, so faults can (and do) occur. This message passing excessively uses function pointers, which breaks stack analysis. So the automatic stack sizing from TinyThread no longer works. That being said, it is now the default threading platform in TinyOS. In Chapter 6 we fix its problems using `UnStacked C with LP`.

Y-Threads [56] is a lightweight threading system which attempts to break each thread into two separate stacks. The first stack is the blocking portion of the thread, and the second part is the non-blocking or shared version of the stack. Since the shared portion of the program does not block, no shared stack storage is required. This inspired the *blocking* attribute in `UnStacked C` found in Chapter 6. In contrast with `UnStacked C`, Programmers must select which portions of the program are blocking (and therefore require stackspace).

Shared-stack cooperative threads [32] are particularly close in spirit to `UnStacked C`. Shared-stack threads operate exactly like regular cooperative threads, except all threads execute on the same system stack. When a thread blocks, it pushes all the

registers onto the stack, then copies that stack to elsewhere. To resume a thread, its stack gets copied back into the system stack, then the registers get popped and execution continues. This means that any pointers to automatic variables will not be valid unless a thread is running. `UnStacked C` functions in a similar fashion in terms of only storing the blocking portion of the continuation, but without the need to copy the stack and without the explicit register operations.

TinyVT [65] is a TinyOS extension that includes a cooperative thread to event compiler. TinyVT is designed to allow users to write their TinyOS event-driven code, in a procedural fashion. It enables users to write a single function with multiple wait statements and have it be transformed into multiple events. Similar to Protothreads, this does not enable building of functional primitives nor hiding the event-driven system, but it does allow procedural programming.

Some preliminary research has been done by Bernauer et al. to convert threads into events [9, 8]. Their work is based upon the idea that cooperative threads can be translated into events. They [8] translated, by hand, some cooperative threads into event-driven code. In their more recent work [9], they explain more details of their translation by hand. They talk about how they plan to make a tool like TinyVT, except it can work off of C code instead of nesC. This is similar to `UnStacked C`, in that they are building up nested structures mirroring call graphs. It differs in that they are forcing all of the once local variables to be global variables, and making only a single instance of each thread possible. This means that dynamic threads will not be possible, nor will multiple instances of the same thread. As of this publication, it is only a concept with no automatic transformation produced. Also similarly to TinyVT they are forcing developers to implement an event-driven set of routines for each blocking routine. Also they only have the capability of making cooperative threads, since they lack Lazy Preemption.

t-kernel [33] is another operating system for wireless sensor networks. It pro-

vides virtual memory and preemptive scheduling on a microcontroller. It accomlishes this through a binary translation, which replaces all of the branch operations and many memory accesses with calls into the kernel – so it can validate operations prior to executing. This allows *t-kernel* to move pages of memory in and out of flash as needed. It also allows *t-kernel* to detect pointer errors, invalid instructions and bad array indexes.

Based on those features alone, *t-kernel* appears to solve many problems, but it does so at a cost. First off it shares the stack between the application and the kernel itself, meaning that multiple threads are not possible. Applications must be specifically written for *t-kernel*, and they will take significantly (roughly 2-2.5 times) longer to execute. The preemption they speak of is a timer they use to detect a failed application. This will then preempt the current application (forcing an exit) and letting the kernel execute. This is accomplished through all of the kernel calls inserted throughout the application. This is similar to the implementation of Lazy Preemption where we insert tests for preemption inside of loops at the source level. *t-kernel* performs a much more in-depth modification, and performs its modification on the binary.

3.2.3 Multi-Thread Scheduling in Real-Time Systems

There are two main categories of multithread schedulers, namely cooperative and preemptive scheduling. To simplify the multithread scheduling landscape we will start by only addressing fixed priority scheduling. Fixed priority schedulers are multithread schedulers in which the priority of a given thread is constant for the life of the thread [49]. A scheduler is said to be preemptive if a thread of a can be preempted or task switched with a different thread. So the main body of research describes this type of scheduler to be a Fixed Priority Preemptive Scheduling (FPPS) [49].

FPPS, or preemption in general, can create a series of problems. Lee et al.

have shown that preemption creates unpredictable temporal variations when used in systems with cache [48]. There has been significant research upon the impact on cache systems [60, 61]. Mutual exclusion is not guaranteed, so additional protections must be added [79]. It can also create problems with timing, since the execution can be interrupted.

On the opposite side of the fixed priority scheduling spectrum, we have Fixed Priority Non-preemptive Scheduling (FPNS) [47]. FPNS runs each task to completion. One such example is TinyOS [35]. Since only one FPNS task can be executing at a time, one stack can be shared. This naturally gives each task exclusive access to all resources, but there is no way to allow a task to yield to any other tasks. This reduces schedulability [79]. Schedulability is the ability to schedule tasks according to meeting deadlines.

In between these two extremes there is a middle ground, namely Fixed Priority Deferred Preemption Scheduling (FPDS). Burn et al. proposed FPDS as threads that will execute exclusively until they "voluntarily suspend" [14]. Burn goes on to call this *cooperative threading*, since instead of forcibly preempting a thread it relies on the thread to yield on its own [13].

FPDS has been researched at length and it treats each thread as a group of multiple non-preemptive subjobs [11]. Each of these jobs executes until completion, then it *yields* to the scheduler. Holenderski et al. recognized that "FPDS is a generalization of FPPS and FPNS, where FPPS can be modeled by FPDS with arbitrarily short subjobs ... and FPNS by FPDS with tasks consisting of a single subjob" [36]. One important component of FPDS is that the programmer of each task (or thread) must manually add in these "preemption points".

FPDS, or cooperative threading as we refer to it, requires developers to insert these preemption points to control the granularity. Klues et al. showed how much cooperative threading can slow down an application [44]. They inserted yield points

throughout a compression routine, and found if a yield is called every iteration of every loop, the computation takes 213% longer. They then tuned it by having it yield every n iterations, and could find a good point. They argue that the need to tune each task to each platform unneccesarily burdens the programmer.

This problem has been solved by Holenderski et al. [36]. They offer FPDS with *optional preemption* points. These are implemented by checking a flag, and if it is set, then yield. These allow developers to manually mark each iteration of a loop with an optional preemption point. This gains the schedulability of preemptive threads without the risks. Bergsma et al. implements this work in the real-time Linux kernel [7].

These *optional preemption* points are the core of Lazy Preemption. Instead of putting the onus on the programmer to insert these optional points, we perform these at the compiler level inside of a source to source translation.

CHAPTER IV

C-XML-C

Compilers are known for their complexity. The theory and steps needed to compile software is typically taught to Computer Science undergraduate students, but few write any modifications or improvements to compilers. Some might attribute this to a lack of education or understanding, but we attribute this to a lack of a framework to develop and modify compilers.

Take for example text editors made for software development. Several editors exist with syntax highlighting and source checking capabilities. Many include plugin frameworks and other extension capabilities [70]. The ideas of extensibility allows new ideas to be evaluated without having to understand and reproduce much of the existing codebase. Many times, when extensions have become popular enough, the text editor adds the functionality from the extensions [57]. That process allows text editors to rapidly evolve new features and capabilities without any risk.

The evolution of text editors is in stark contrast to how slowly compilers change. Compilers and their respective programming languages evolve slowly, or at least, only in large steps. The evolutions in programming languages can be seen when an entirely new language develops. One of the reasons this is common is that

it takes so much work to implement a compiler, making one small change does not justify the time.

This work focuses on the C language, since it is prevalent in many different systems and has a complex standard which defines it. We present a complete framework to add modifications to the C language, without forcing developers to learn a specific language and without having to learn an API. We do this by providing a compiler that translates C source code into XML and then back into C source code. This allows developers to run the compiler, manipulate the resulting XML and then produce modified source code on the output.

4.1 Implementation

C-XML-C contains three main components, a compile manager which controls the order and execution of all other components, a parser which translates the code into XML and a generator which translates the XML back into source code. The compile manager controls the compilation process and accepts gcc style arguments, allowing it to replace gcc in existing makefiles. The parser is CIL patched so it outputs the file data to an XML file [51]. We also applied merged patches from Cooprider et al. that adds supports for TinyOS and embedded systems in general [16]. The generator is written in Python and performs the XML to C translation. It is called by the compile manager after any required transforms are complete.

4.1.1 Compile Manager

The compile manager uses CIL to merge all the files into a single file for transformation and then compilation. It is flexible enough to allow multiple transformations to be strung together in any order. It ties the other components together through a configuration file. This configuration file controls the compilation process.

```
1  [Pass1]
2  command = stackswap.py functionlist.txt %1 %2
3  debug = True
4  [Pass2]
5  command = threadsafe.py functionlist.txt %1 %2
6  clean = True
7
8  [Output]
9  #Command in the output is the compiler to be used
10 command = gcc -O3 -fomit-frame-pointer
11 #platform is the c-xml to be used (i.e. native, avr or msp430)
12 platform = native
13 #cxmlc is an optional argument to the base path to cxmlc
14 path = ../../
```

Listing 18: Config.ini: An example configuration file for the Compile Manager

The configuration files offer flexibility in the compilation process. They are INI files which can contain not only the configuration but also comments.

As shown in Listing 18, there are two main portions: the `output` and the `passes`. There are two passes in this configuration file (shown on lines 1 through 6). Each pass must have a command string. This string is the command line transform to be executed. Notice on lines 2 and 5 the command strings contain a % 1 and a %2. These are replaced by the input file and the output files respectively. The other options for each pass are `debug` and `clean`. `Debug` writes the standard out of the transform, and `clean` is explained below.

The `output` section controls the output command. The output `command` is the command-line program, usually a compiler, used for preprocessing and for the final compilation. It also includes the platform, which selects which C-XML transform because the C-XML transforms differ slightly when cross-compiling[1] for different platforms. Included is an 8-bit (avr), a 16-bit (msp430) and either 32 or 64-bit based upon the machine compiling. The path string should represent the path to the C-XML-C installation, which contains the C-XML and XML-C programs.

The Compile Manager runs the preprocessor on each individual C file prior to merging. Next it converts the merged C files into XML. It then executes the passes

[1]Cross-compiling is compiling an application to run on a different platform than your compiler is running on.

in order. Since a previous transform may have manipulated the XML file outside of the normal structure, some transforms require a clean or unmodified XML file. Clean XML files maybe required in case two sequential transforms do not interact poorly. This is controlled by the `clean` flag as shown on line 6 of Listing 18. A detailed flowchart of this algorithm is shown in Figure 13.

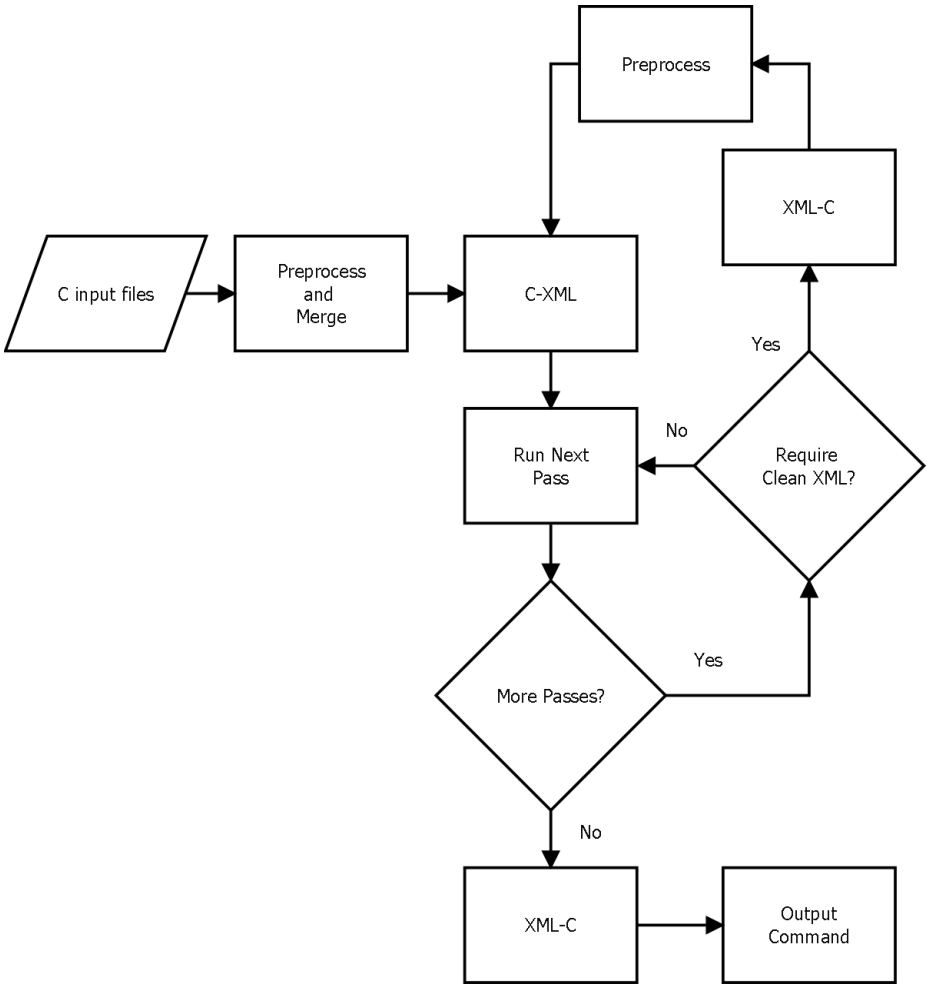


Figure 13: This flowchart shows the Compile Managers execution.

4.1.2 Parser

CIL already includes many features which our parser inherits. Since CIL can parse both GCC specific and MSVC specific code so can C-XML-C.[2] We use GCC's preprocessor to preprocess the source code. Merging the program prior to linking allows C-XML-C to combine multiple C files into a single XML file. This enables *whole program transformations*.

CIL already simplifies C code into a subset of C so that the code is uniform [54]. For example, all looping and conditionals are converted into a consistent pattern. All of these features are inherited by the parser.

```
1  int main(){
2    int i;
3    int k=0;
4    for(i=0;i<10;i++){
5      k++;
6    }
7    i=0;
8    while(i<10){
9      k++;
10     i++;
11   }
12   return 0;
13 }
```

Listing 19: Input.c: An example C file with two simple loops

```
1  <statement>
2    <infiniteLoop>
3      <block>
4        <statement>
5          <if>
6            <guard>
7              <expression kind="binaryOp" operator="lessThan">
8                <expression kind="lvalue">
9                  <lvalue>
10                   <base kind="variable">
11                     <variableUse name="i" id="140" isGlobal="false" isAddressTaken="false"
                             isUsed="true"/>
12                   </base>
13                 </lvalue>
14               </expression>
15               <expression kind="constant">
16                 <constant kind="int">
17                   <value>10</value>
18                 </constant>
19               </expression>
```

[2]We have only tested C-XML-C with GCC. Minor modifications to the Compile Manager would need to be made to support MSVC style arguments on the command line, but the core C-XML, XML-C and the transforms would not have to be changed.

```
20              <type kind="int"/>
21            </expression>
22          </guard>
23          <thenBranch>
24            <block/>
25          </thenBranch>
26          <elseBranch>
27            <block>
28              <statement>
29                <break>
30                  <location file="example.c" line="4" byte="34"/>
31                </break>
32              </statement>
33            </block>
34          </elseBranch>
35          <location file="example.c" line="4" byte="34"/>
36        </if>
37      </statement>
38      <statement>
39    -- Skipped 46 lines --
40      </statement>
41    </block>
42    <location file="example.c" line="4" byte="34"/>
43  </infiniteLoop>
44 </statement>
```

Listing 20: Output.xml: An excerpt from an XML AST from the simple C example

For example, Listing 19 contains an input file with two loops. The first loop is a `for` loop and the second is a `while` loop. In Listing 21 we can see that both of these loops become identical in the transformed C code. This is one of the many simplifications that CIL provides. The actual XML AST of the loop itself can be seen in Listing 20. Both loops are exactly identical with the exception of different line numbers in the location tags.

```
1  /*printout =*/
2  int main(){
3    int i;
4    int k;
5    {
6      k = 0;
7      i = 0;
8      for(;;){
9        {
10         if(((i) < (10))){
11         }else{{
12           break;
13         }
14         }
15         k = ((k) + (1));
16         i = ((i) + (1));
17       }
18     }
19     i = 0;
20     for(;;){
21       {
22         if(((i) < (10))){
23         }else{{
```

```
24            break;
25          }}
26          k = ((k) + (1));
27          i = ((i) + (1));
28        }
29      }
30      return 0;
31    }
32  }
```

Listing 21: Output.c: An example C file after it is passed through C-XML-C. The indenting was fixed manually for presentation.

4.1.3 XML-C

XML-C processes a given XML tree, then generates valid source code. It operates as a visitor on the XML AST building C source code output as it runs. It is written in Python and can be easily extended to support new types of tags. XML-C maps each XML object type to a specific function (of the same name). All of these functions are registered in a large dispatch table which executes the correct one based upon the incoming node in the AST. This allows new objects to be added if a transform requires it. [3]

4.2 Usage

In most cases, developers need to setup the config file, choose or write a transformation, and run the compile manager as if it were the compiler. When writing new transforms, developers can run example programs to get example XML. They can use those examples to figure out how to programmatically achieve their transformation.

It is important to note that the XML is the complete abstract syntax tree (AST) of the source code. It also includes extra information, such as line and column numbers. The XML to C translation does require that the input files be valid XML,

[3]None of the transforms that we implemented required any additional object types added to XML-C, but it is trivial to do so. This could be done to support modifications to C itself, or possibly other extensions to translate other languages into C. These are outside of the focus of our work, but possible in the framework.

but it does not check the validity of the AST. It will still generate source code, but the source code may not be valid.

This lack of error checking enables developers to shortcut XML tags to assist in transformations. We added an XML tag that has not been shown in previous examples that developers can use to simplify their code generation. In other source-to-source translators, to insert blocks of code into an AST, developers have to regenerate a valid portion abstract syntax tree. Instead of having to regenerate new portions of the AST, with C-XML-C they can insert a *PRE* tag, which signifies preformatted output. The text in these blocks will pass directly to the output.

```
1  <CBody>
2    <Types>
3      <Type id="1" name="int"/>
4    </Types>
5    <PRE>
6      int error_state=0;
7    </PRE>
8    <FunctionDecl column="5" line="13" returntype="1">
9      <FunctionArgs column="13" line="13"/>
10     <FunctionBody column="15" line="13">
11       <PRE>
12         {if(error_state&gt;0){printf("Running␣main␣with␣error_state␣in␣%d\n",error_state)
               }}
13       </PRE>
14       <Statement>
15         <FunctionCall column="4" line="14">
16           printf
17           <FunctionArgs column="10" line="14"/>
18             <TEXT column="11" line="14">
19               &quot;Hello World.\n&quot;
20             </TEXT>
21           </FunctionArgs>
22         </FunctionCall>
23         <SEMICOLON column="28" line="14">
24           ;
25         </SEMICOLON>
26       </Statement>
27       <ReturnStatement column="4" line="15">
28         return
29           <NUMBER column="11" line="15">
30             1
31           </NUMBER>
32       </ReturnStatement>
33     </FunctionBody>
34   </FunctionDecl>
35 </CBody>
```

Listing 22: XML file after top level statements have been parsed

Listing 22 shows a PRE tag on lines 11-13 which shows what would normally require the developer to create over a dozen XML nodes to replace it. Instead devel-

opers can insert PRE tags. This allows developers to ignore the details of the AST itself, and focus on what is important to their modifications.

4.3 Transforms

Since C-XML-C uses XML to store its ASTs, transforms can be implemented in any language that can read and write files. There are many different APIs for accessing XML files. One such API that most programming languages support is called the Document Object Model (DOM) [3]. The DOM is a consistent API for accessing, manipulating and creating XML documents. In the following transforms, we leverage this API for manipulating the ASTs of C programs. We provide several example transforms in several different programming languages.

In this dissertation we present eight different transforms. Most were written in Python, but others were written in eXtensible Stylesheet Language Transformations (XSLT) [40], Java and C#. A list of the transforms, the language of implementation and the number of lines of code can be found in Table II.

4.3.1 XSL Callgraph

Deriving the callgraph of a program can be very useful. It can show direct and indirect recursion. It can also uncover hidden source code dependencies. A callgraph is a graph where each function is a node. There is a directional arrow to the callee from the caller. Call graphs are explained in detail in Section 2.2. A call graph is a common requirement for many more complex transformations and code analysis.

```
1 void b(){
2 }
3 void c(){
4 }
5 void a(){
6   b();
7 }
8 void d(){
9   a();
```

Transform Name	Programming Language	Lines of Code
Callgraph	XSLT	14
Tail Recursion	Python	170
Volatile Fix	C#	350
Forced Function Inlining	Java	300
Thread Safety	Python	200
Stack Swap	Python	250
Indirect Call Removal	Python	400
`UnStacked C with LP`	Python	500

Table II: List of all included transforms.

```
10   b();
11   c();
12   d();
13 }
14 void main(){
15   d();
16   a();
17 }
```

Listing 23: Example program for Callgraph analysis

Since the intermediate source code is stored as XML, we can use Extensible Stylesheet Language (XSL) to implement a transform. Listing 23 shows a simple C program which we can run through C-XML-C and it generates a seven kilobyte XML file. After running through the XSL transform shown in Listing 24 it generates an output shown in Listing 25. The output XML file can easily be parsed and loaded by any application. It also can be humanly read.

```
1  <?xml version="1.0" encoding="utf-8" ?>
2  <xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
3    <xsl:template match="/">
4      <CallGraph>
5        <xsl:for-each select="//functionDefinition">
6          <xsl:element name="{@name}">
7            <xsl:for-each select="descendant::functionCall/name/expression/lvalue/base/
                 variableUse">
8              <xsl:element name="{@name}" />
9            </xsl:for-each>
10         </xsl:element>
11       </xsl:for-each>
12     </CallGraph>
13   </xsl:template>
14 </xsl:stylesheet>
```

Listing 24: Callgraph Extensible Stylesheet code

```
1 <?xml version="1.0"?>
```

```
2  <CallGraph>
3    <b/>
4    <c/>
5    <a>
6      <b/>
7    </a>
8    <d>
9      <a/>
10     <b/>
11     <c/>
12     <d/>
13   </d>
14   <main>
15     <d/>
16     <a/>
17   </main>
18 </CallGraph>
```

Listing 25: Callgraph output

XSL is a declarative language that takes XML files as its input and outputs a single XML file. XSL transforms are XML documents themselves as shown in Listing 24. Most XSL transforms start by matching the root document as shown in Lines 3 and 13 of Listing 24. Nodes which do not start with "xsl" are passed directly to the output. We use this to generate the new root object named `CallGraph` on Lines 4 and 12 in Listing 24. Then we use a for loop to iterate through all the XML objects that are of type `functionDefinition` on Line 5. For each of these objects, we create an element which takes the name of the object (a `functionDefinition`) attribute named `name` on Line 6. Inside of each `functionDefinition` we search for a specific pattern of objects named `functionCall, name, expression, lvalue, base, variableUse` on Line 7. This pattern matches the XML form of a direct function call. We then generate an XML object off of the `variableUse`'s attribute named `name` on Line 8, which forms the links in our call graph.

The XSL Callgraph transform is the only example that does not modify the output. All of the other transforms in this chapter modify the output XML file so that it changes the way the source code compiles. This transform can be used as a building block for other transforms.

4.3.2 Tail Recursion

Tail recursion optimization is a common optimization many compilers can perform. A tail recursive function is one that calls itself only as the last operation performed. Since the function is only recursive at the end of the routine, the previous stack frame is not required and can be replaced with the next stack frame. Compilers often implement this optimization, since the only operation that needs to change is that instead of a `call` instruction, simply `goto` the beginning of the routine instead.

```
1  void factorial_i(double number, double * product){
2    if(number <= 1){
3      return;
4    }else{
5      *product *= number;
6      factorial_i(number - 1, product);
7    }
8  }
9
10 double factorial(double number){
11   double result = 1;
12   factorial_i(number, &result);
13   return result;
14 }
```

Listing 26: Example of tail recursion before any optimizations

Listing 26 shows a proper tail recursion [15] implementation of a factorial. For proper tail recursion, a return value cannot be used. Instead of using a return value, a product argument is used. Factorial_i performs tail recursion since it only calls itself at the end of itself.

```
1  void factorial_i(double number, double * product){
2  start:
3    if(number <= 1){
4      return;
5    }else{
6      *product *= number;
7      number = number - 1;
8      product = product;
9      goto start;
10   }
11 }
12
13 double factorial(double number){
14   double result = 1;
15   factorial_i(number, &result);
16   return result;
17 }
```

Listing 27: Example of tail recursion optimization

Listing 27 shows the output of the transformed source code. Notice the label added on line 2. Lines 7, 8 and 9 replace the originally recursive call. Lines 7 and 8 overwrite the arguments with new argument values and Line 9 branches back to the beginning.

We implemented the tail recursion transform in about 170 lines of python code. No libraries outside of the python standard libraries were required. It is implemented as a simple two pass process using a visitor [28]. The first pass checks to find which routines implement tail recursion. The second pass performs the transformation.

Our visitor object executes in a recursive fashion. It calls an `enter` method as it enters each node, prior to processing any of the child nodes. It then calls an `leave` method after processing all of the child nodes. The visitor that implements the first pass is shown in Listing 28.

It performs a set of pattern matching as it is called on each node. It performs this pattern matching as it transitions into different states as it finds specific types of nodes. In the `leave` method, it restores the previous states, so that the state machine can naturally unwind on lines 36-42 of Listing 28. The trickiest part of the whole algorithm is detecting tail recursion versus regular recursion. After a function is found to be recursive in lines 28 and 29, the state is set to `CHECKING`. All the checking state does is look for `instruction` objects. If any `instruction` objects are found, then the call is not the last instruction in the function, and it is not tail recursive (lines 55-58).

```
1  class SearchVisitor(visitor):
2    IDLE, INFUNCTION, INBODY, INCALL, INNAME, CHECKING = range(6)#FSM for search
3    def __init__(self):
4      self.state = self.IDLE #Current state of search
5      self.lastworking = []  #Stack of name of searching function (gcc extension)
6      self.last = [] #Stack of previous state values
7      self.working = "" #name of current function
8      self.calls = [] #List of recursive functions
9    def enter(self, node):
10     self.last.append(self.state) #Allow the state to be restored upon leaving
11     #this fixes problems if a child function is defined inside of another (gcc extension)
12     self.lastworking.append(self.working)
13     if(self.state == self.IDLE): #looking for a function
14       if(node.nodeName == "functionDefinition"):
```

```
15          self.working = node.getAttribute("name")
16          self.state = self.INFUNCTION #Entering a function
17        elif(self.state == self.INFUNCTION):
18          if(node.nodeName == "result"): #Check for return value
19            if(node.getElementsByTagName("type")[0].getAttribute("kind") != "void"):
20              self.state = self.IDLE #basically if the function does not return void, then
                    ignore
21          if(node.nodeName == "functionCall"):
22            self.state = self.INCALL #We found a function call
23        elif(self.state == self.INCALL):
24          if(node.nodeName == "name"):
25            self.state = self.INNAME #We found the name of the called function
26        elif(self.state == self.INNAME):
27          if(node.nodeName == "variableUse"):
28            if(self.working  == node.getAttribute("name")):
29              self.calls += [self.working]#Found a recursive function
30              self.state = self.CHECKING
31        elif(self.state == self.CHECKING):
32          if(node.nodeName == "instruction"): #recursive call is not the last instruction
33            print "%s is not proper tail recursion :(" % self.working
34            if(self.working in self.calls):
35              self.calls.remove(self.working)
36      def leave(self, node):
37        #restore the last state
38        laststate = self.last.pop()
39        if(self.state == self.CHECKING):
40          if(laststate == self.IDLE):
41            self.state = laststate
42        self.working = self.lastworking.pop()
```

Listing 28: Implementation of first pass of a visitor searching for tail recursive functions

4.3.3 Volatile Fix

The next simple transform marks global variables as `volatile` if they are accessed by the main loop and inside of an interrupt. It also issues a list of all shared variables. In embedded systems, if a global variable is written in the main loop and then accessed in an interrupt, a data fault can occur. The keywork `volatile` forces each access to actually read/write RAM as opposed to leaving them in registers [39]. The `volatile` keyword alone does not fix possible shared data faults in preemptive threading systems, but the list of shared variables it prints out can lead developers toward finding and fixing possible faults.

Listing 29 shows how a variable, whose operations are already atomic, can have an incorrect output in the main loop. In that example, it is possible that the main loop never reads the variable as one, since it is cached in a register. The `volatile`

keyword does not make the variable accesses atomic, so other considerations for interrupt accesses are required. For this example we only focused on the `volatile` keyword.

```
1  void factorial_i(double number, double * product){
2  start:
3    if(number <= 1){
4      return;
5    }else{
6      *product *= number;
7      number = number - 1;
8      product = product;
9      goto start;
10   }
11 }
12
13 double factorial(double number){
14   double result = 1;
15   factorial_i(number, &result);
16   return result;
17 }
```

Listing 29: This source code shows how a possible data fault can occur as the result of an interrupt if the flag is not marked `volatile`

Our volatile fix transform is implemented in C#. It only uses the standard C# libraries in about 350 lines of source code. It is implemented in two passes with a visitor. The first pass detects all of the possible volatile faults and the second pass adds the volatile keywords.

4.3.4 Force Function Inlining

Functions can be inlined by compilers so that they are expanded as if they were preprocessor macros. C includes the keyword `inline` to enable the compiler to inline a function, but the inlining of the function is not guaranteed. Instead of relying on a compiler that may or may not force inline a given routine, we implemented a transform that performs forced function inlining. Listing 30 shows a simple `increment` routine that calls an `add` routine. Listing 31 shows the inlined expansion of the `add` routine.

```
1 inline int add(int a, int b){
2   return a+b;
3 }
4
5 int increment(int x){
```

```
6   return add(x,1);
7 }
```

Listing 30: A simple incrementing routine

Our transform performs two passes on the source code. The first pass uses a visitor to search through the source code for the `inline` keyword while simultaneously building a call graph. The second pass processes the call graph to replace any instances of the inlined functions with an equivalent expansion.

```
1  int increment(int x){
2    return ({
3        int inlined_retval1;
4        int inlined_a; int inlined_b;
5        inlined_a = a;
6        inlined_b = 1;
7        inlined_retval1 = inlined_a + inlined_b;
8        inlined_retval1;
9        });
10 }
```

Listing 31: Example of force inlined incrementing routine

Notice in Listing 31, that the outputted source code is valid, but it may confuse a following transform. Instead, a developer would want to clean the XML file prior to applying any other transform. This would force the expression-statement[4] to be broken into multiple sequential statements.

This transform was implemented in 300 lines of Java. It uses only standard class libraries, without any external dependencies.

4.3.5 Thread Safety

In multithreaded programming, thread safe code is code that operates correctly when called simultaneously from multiple threads, or contexts of execution. Thread safety is not only an issue with a single routine being executed from multiple threads simultaneously, but also a library where multiple calls may interact with each other. Unless a library is written specifically with thread safety in mind, it is likely not thread safe.

[4] An expression-statement is a statement that takes place inside of an expression.

If it uses any global variables, it is likely not thread safe. To help with this problem we have written a transform that can fix many common thread safety issues.

If a library is not thread-safe, than it can only have one thread calling into it at a time. One way to create that safety requirement is to lock a mutex prior to accessing any of the library routines and then unlocking the mutex after each library routine exits. When a second thread attempts to lock an already locked mutex, it must block until the mutex becomes unlocked. This is a common method of making sure that only a single thread can execute any of the libraries' functions at a time.

Our thread safety transform takes a group of functions (i.e. a library) and makes them thread safe. It make a wrapper function for each one, that wraps the functions in a single mutex so that the functions cannot execute concurrently. This transform can be run in two different ways. The first way uses a single global mutex to protect all of the routines from executing concurrently as shown in Listing 33, as transformed from the original routines shown in Listing 32.

```
1 int add(int a, int b){
2   return a+b;
3 }
4
5 int sub(int a, int b){
6   return a-b;
7 }
```

Listing 32: A pair of simple routines

In line 6 of Listing 33 a global mutex is created and initialized. The original functions `add` and `sub` (Listing 32 lines 1 and 5) have been renamed to `thread_safe_add` and `thread_safe_sub` (Listing 33 lines 1 and 15) respectively. It then generates a new `add` in lines 7-13 of Listing 33. This new `add` locks the mutex (line 9), calls `thread_safe_add` (line 10), unlocks the mutex (line 11) and returns the proper value (line 12). A similar method is used to implement the new `sub` routine in lines19-25.

```
1 int thread_safe_add( int a,  int b){
2   return a + b;
```

```
3 }
4
5 #include <pthread.h>
6 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
7 int add(int a, int b){
8   int safe_return;
9   pthread_mutex_lock(&mutex);
10  safe_return =   thread_safe_add(a,b  );
11  pthread_mutex_unlock(&mutex);
12  return safe_return;
13 }
14
15 int thread_safe_sub( int a,  int b){
16  return a - b;
17 }
18
19 int sub(int a, int b){
20  int safe_return;
21  pthread_mutex_lock(&mutex);
22  safe_return =   thread_safe_sub(a,b  );
23  pthread_mutex_unlock(&mutex);
24  return safe_return;
25 }
```

Listing 33: Example of a global mutex protecting a pair of routines from multiple threads executing the routines simultaneously

Another way to run this transform is to make multiple functions allowed to be run simultaneously, but only one instance of each function may be executed as shown in Listing 34. The noticable differences between the global mutex and the single mutex is that each function requires its own mutex as shown in lines 8 and 21 of Listing 34. The rest of the code remains unchanged. In this mode, multiple functions can be executed simultaneously, but only a single instance of each function can occur simultaneously.

```
1 int thread_safe_add( int a,  int b){
2   return a + b;
3 }
4
5 #include <pthread.h>
6 int add(int a, int b){
7   int stack_return;
8   static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
9   pthread_mutex_lock(&mutex);
10  stack_return =   thread_safe_add(a,b  );
11  pthread_mutex_unlock(&mutex);
12  return stack_return;
13 }
14
15 int thread_safe_sub( int a,  int b){
16  return a - b;
17 }
18
19 nt sub(int a, int b){
20  int stack_return;
21  static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
22   pthread_mutex_lock(&mutex);
23   stack_return =   thread_safe_sub(a,b  );
24   pthread_mutex_unlock(&mutex);
25   return stack_return;
26 }
```

Listing 34: Example of a mutex for each routine protecting against multiple points of execution running the same function simultaneously

This enables embedded developers to integrate libraries into their multithreaded system without worrying about thread safety. This also avoids the programmer of performing this same task manually.

This transform is about 200 lines of Python source code. It is configurable on the command line which mode to run in, and takes an argument for the file name containing a list of functions to make thread safe. This allows users to simply change a configuration file and run on any application. This transform is done in a single pass through the XML AST.

4.3.6 Stack Swap

Another transform which is structurally similar to thread safety is stack swapping. We define stack swapping as a single thread running and switching to a different stack for a single call to a function, then swapping stacks back after the function returns. This is important for library vendors, since they cannot guarantee that an application's compiler will have allocated enough stack space for the given library. This is especially important when a library may need to make recursive calls or calls into other libraries with large stack requirements. Often embedded systems interact with PCs. They often communicate with custom drivers or libraries. While many programs are compiled with modern compilers, some programs are written in an alternate fashion such as with LabVIEW. These alternate programming tools may not necessarily allocate a large enough stack space [38].

This transform wraps functions similarly to the thread safety transform. It creates a wrapper for each function that allocates and switches to a separate thread

prior to executing. It also copies all of the arguments into a global structure so that they can be accessed from the new stack. The resulting transform is not thread safe, but can be made so using the thread safety transform. Listing 35 shows the source code prior to transform and Listing 36 shows output of the transforms.

```
int add(int a, int b){
  return a+b;
}
```

Listing 35: A simple routine prior to stack swapping

```
struct stack_struct_add{
  int stack_return;
  int a;
  int b;
}stack_struct_add_v;

  void stack_swapped_add(){
  {
    stack_struct_add_v.stack_return = ((stack_struct_add_v .a) + (stack_struct_add_v .b));
return;
;

  }

}

 int add(int a, int b){

  {
  #include <ucontext.h>
  ucontext_t current, child;
  //Setup the new context
  void * malloc(size_t size);
  void * newstack = malloc(SIGSTKSZ);
  getcontext(&child);
  child.uc_stack.ss_sp = newstack;
  child.uc_stack.ss_size = SIGSTKSZ;
  //Here force it to return to the caller
  child.uc_link = &current;
  makecontext(&child, stack_swapped_add, 0);
  stack_struct_add_v.a = a;
  stack_struct_add_v.b = b;

  //Swap the context
  swapcontext(&current, &child);
  //we return then we are complete
  free(newstack);
  return stack_struct_add_v.stack_return;
  }

}
```

Listing 36: A simple routine swapping to a separate stack prior to executing

The stack swap transform is about 250 lines of Python code. It completes the transform in a single pass of the XML AST, making multiple modifications as it runs.

4.3.7 Indirect Call Removal

This transform removes indirect calls from a given application. This transform groups function pointers together and makes dispatch tables which execute all of the possible values. It replaces function pointers with indexes into these dispatch tables, and it replaces indirect calls to function pointers with these dispatch tables. This allows a more accurate call graph to be represented. This also, in turn, allows other tools such as stack analysis and `UnStacked C` to be used in cases where it normally could not be. It also permits a compiler to inline a function in spite of function pointers.

This transform is broken into two main stages. The first stage detects the additional call graph members from function pointers. The second stage performs the transform on the AST.

Detecting function pointers in a call graph can be done through an analysis of all of the pointers in a system. Milanova et al. [52] implements a simple FA pointer analysis initially proposed by Zhang [81] to complete the call graph in the face of function pointers. FA pointer analysis is flow and context insensitive, and can yield some false positives. We implemented an algorithm similar to the one Milanova proposed. This means that we must build a graph of all of the relations between all of the variables in the system, including all of the function arguments and return values. Once this directional relationship graph has been built, we can start from the function pointer dereferences and work backwards to determine all of the possible values a given pointer can have at any given time.[5]

This transform iterates through all variable accesses and identifies which variables pass values to each other. Each set of variables that can possibly be written to another variable is directionally linked. Then each constant value that is in the right hand side of any equation that writes into one of these variables is noted for that

[5]To achieve this, then entire program's source code must be accessible. This includes standard library functions.

variable and all of the variables it is linked to. This forms a list for each variable, all of the possible constant values. In the case of function pointers, it builds up a list of all possible function pointers a given indirect call could possibly have, and then builds a dispatch table for that specific position.

An example input is shown in Listing 37 has two sets of function pointers. There are two groups of function pointers in the system. We analyze the entire system and the transform built up two dispatch tables and replaced the indirect calls with the dispatch tables. This can be seen in Listing 38. Notice how the types of all of the variables have not been changed, instead we only modify the constant value inputs into the function pointers and add calls to the dispatch tables at the correct locations.

```
1  #include <stdio.h>
2
3  void f1(void){
4    printf("ft1\n");
5  }
6  void f2(void){
7    printf("ft2\n");
8  }
9  void f3(void){
10   printf("ft3\n");
11 }
12 void g1(int x){
13   printf("g1:%d\n",x);
14 }
15 void g2(int x){
16   printf("g2:%d\n",x);
17 }
18 void g3(int x){
19   printf("g3:%d\n",x);
20 }
21 typedef void (*func1)(void);
22 typedef void (*func2)(int);
23
24 func1 hide(func1 x){
25   return x;
26 }
27
28 func1 list1[10];
29 int list1count = 0;
30 func2 list2[10];
31 int list2count = 0;
32
33 void post1(func1 f){
34   list1[list1count++] = f;
35 }
36 void post2(func2 f){
37   list2[list2count++] = f;
38 }
39
```

```
40 int main(){
41   printf("Hello...\n");
42   post1(f1);
43   post1(hide(hide(hide(f2))));
44   post1(f3);
45   post2(g1);
46   post2(g2);
47   post2(g3);
48
49   while(list1count--){
50     (*list1[list1count])();
51   }
52   while(list2count--){
53     (*list2[list2count])(list2count);
54   }
55   printf("done...\n");
56   return 0;
57 }
```

Listing 37: Input file which contains two pools of function pointers

```
1  void dispatch_834 (int dispatcher){
2      switch (dispatcher){
3        case 1:
4      f3 ();
5      break;
6        case 4:
7      f1 ();
8      break;
9        default:          /* case 2: */
10     f2 ();
11     break;
12     }
13 }
14
15 void dispatch_836 (int dispatcher, int x){
16     switch (dispatcher){
17       case 3:
18     g3 (x);
19     break;
20       case 5:
21     g1 (x);
22     break;
23       default:          /* case 6: */
24     g2 (x);
25     break;
26       }
27 }
28
29 int main (){
30   func1 tmp;
31   func1 tmp___0;
32   func1 tmp___1;
33   int tmp___2;
34   int tmp___3;
35   {
36     printf ("Hello...\n");
37     post1 ((void *) 4);
38     tmp = hide ((void *) 2);
39     tmp___0 = hide (tmp);
40     tmp___1 = hide (tmp___0);
41     post1 (tmp___1);
42     post1 ((void *) 1);
43     post2 ((void *) 5);
44     post2 ((void *) 6);
45     post2 ((void *) 3);
```

```
46      for (;;){
47        tmp___2 = list1count;
48        list1count = ((list1count) - (1));
49        if (!tmp___2){
50          break;
51        }
52        dispatch_834 ((unsigned int) (*(list1[list1count])));
53      }
54      for (;;){
55        tmp___3 = list2count;
56        list2count = ((list2count) - (1));
57        if (!tmp___3){
58          break;
59        }
60        dispatch_836 ((unsigned int) (*(list2[list2count])), list2count);
61
62      }
63      printf ("done...\n");
64      return 0;
65    }
66  }
```

Listing 38: Output file which contains no function pointers

We have successfully tested this transform on large scale TinyOS applications to remove the indirection caused by threads and its message passing interfaces. This allows stack analysis to be run on these threaded TinyOS systems. It also allows the indirect calls to be inlined.

4.3.8 Other Transforms

The other transform, `UnStacked C with LP`, will be discussed in detail in Chapters 5 and 6. It is a complex transformation that translates multithreaded source code into event-driven state machines. Its implementation is more complicated since it modifies not only variable allocates, but also all function invocations, signatures and bodies.

4.4 Compiler Test

There exists a body of source code which is known to break compilers. This test suite is known as the GCC C-Torture Test Suite [67]. This has been built by people submitting source code that is known to break compilers. C-XML-C passed 34329

tests out of more than 60,000 tests. GCC[6] itself only passed 45086 of the tests. The reason C-XML-C failed for many of the tests is because many of the tests attempted to use C-XML-C for partial compilation as opposed to whole-program compilation (which is the only method of compilation it supports). In any event, passing 34329 tests is a strong indication that many different types of C source code will compile correctly inside of C-XML-C.

4.5 Summary

C-XML-C is a compiler framework for creating whole-program C compiler extensions in any language. We have implemented many different transformations which are useful to embedded system developers. Thread safety, stack swapping, call graph analysis and volatile fix can assist embedded system developers in making their system simpler and safer. Tail recursion, force function inlining and indirect call removal can perform different optimizations which are useful to embedded system developers.

These diverse and complex transforms show the flexibility of the C-XML-C framework. In contrast, the CIL paper itself only talks about three different transforms it performs on the source code. Instead we offer eight transforms. Each of these was implemented in less than a day of work, with the exception of `UnStacked C` and Lazy Preemption. Since these transforms can be written in any programming language, developers can quickly make translations in the language they are most comfortable – enabling more experimentation and exploration in compilers.

[6]Tests were run with GCC version 4.4.5 compiled for x86_64-linux-gnu.

CHAPTER V

UnStacked C

Multi-threaded programming and event-driven programming are the popular approaches for building concurrent systems. While there are several instances in the literature where the two approaches have been presented as opposing forces, Adya et al [1] provide a nice treatment of identifying the essential differences. The most important difference, it turns out, is the way the context execution stacks are managed: manual in the case of event-driven programming, and automatic in the case of threaded programming.

The two approaches have distinct advantages and disadvantages. Event-driven programs tend to be very efficient in terms of memory footprint. However, *writing* event-driven programs is hard: the programmer has to manually *rip* the stack, and maintain state across multiple tasks. Because the programmer has to manually deal with maintaining state across tasks, each individual task does not need to maintain its own stack. Multi-threaded programs, by contrast, are easier to write and comprehend. The program is expressed as a sequence of actions, without regard for implicit points where one task may yield to another. The stack is managed automatically by the thread scheduler. The big disadvantage with this approach is that the memory

requirements of threaded programs are usually much larger than their event-driven counterparts, since each thread has to maintain its entire context while it is blocked.

Note that when we talk about multithreading in the context of this chapter, we are talking about *cooperative* threads. We will address *preemptive* threads in Chapter 6. Cooperative threads, in contrast with preemptive threads, only yield the processor to other competing threads at *well-defined* points in the execution. These points are either explicitly defined by the programmer through a yield, or more commonly by calling some other routine that will block. Common blocking routines are blocking I/O routines or protection primitives (e.g., mutex or semaphore operations).

While the memory overhead of cooperative threads may not be a big problem for PC applications, the overhead does become a significant handicap when implementing software either for embedded systems (where memory is a scarce resource), or for high-concurrency servers that run thousands, even millions, of threads. As it stands, if the amount of available memory is insufficient for the threads that an application will need, the available alternative is to program the system in an event-driven style. Event-driven programs are hard to write, and even harder to read and reason about [74]. It would be nice to enjoy the benefits of threading without its memory overhead.

This is the primary contribution we make in this chapter. We present `UnStacked C`: a source-to-source translator that enables C programmers to write code in a multithreaded fashion, but executes using event-driven semantics. The translator converts regular C code that uses cooperative threads into *tasklets*. Since these tasklets follow event semantics, they do not need to store their stack, and hence their memory overhead is substantially reduced. At the same time, the readability of the program is not compromised: programmers can use most regular C constructs.

Outside of the embedded systems community, some C programmers have looked for ways to get to the "million-thread mark", and thus far, there is no way to achieve

it [34]. Our system enables programmers to write applications using stackless threads, and can: in fact, we reach and exceed that million-thread goal.

5.1 `UnStacked C` Translation Strategy

At its heart, `UnStacked C` is a C-XML-C transform that takes as input C code using cooperative threads, and produces an output code that uses stackless continuations. The continuation-based code is semantically equivalent to the original code, but because individual thread stacks are no longer necessary, it has a much smaller memory footprint than the multithreaded code. `UnStacked C` is based upon three different implementation strategies.

1. Our initial strategy for the continuations in `UnStacked C` is based on *Duff's Device* [18], which is also the strategy used by Protothreads [20] and Tame [45].

2. We also have an operational mode which can generate jump tables directly using goto statements to labels throughout the code, instead of using Duff's Device.

3. We also can use GCC's "labels as value"[68], which is a C extension that allows a pointer to hold an address and to goto it without a call in C.

We will compare these three methods of continuations inside of `UnStacked C`.

5.1.1 Translation Rules

Here, we describe the rules that we use for translating cooperatively-threaded code into stackless continuations. Throughout this section, we will use the code example in Listing 39 to illustrate the rules we describe.

```
1 Echo Server implemented using a thread library:
2 int echo(int port) {
3   int listenfd, *connfdp, clientlen;
4   struct sockaddr_in clientaddr;
5   pthread_t tid;
```

```
6   listenfd = open_listenfd(port);
7   while (1) {
8     clientlen = sizeof(clientaddr);
9     connfdp = malloc(sizeof(int));
10    *connfdp = accept(listenfd,
11                (SA *) &clientaddr, &clientlen);
12    pthread_create(&tid, NULL, thread, connfdp);
13  }
14 }
15
16 Data structure to store context for continuation:
17 typedef struct UnStackecho {
18   uint8_t state;
19   union {
20     struct { int port; } args;
21     int retval;
22   } ops;
23   struct {
24     int listedfd, *connfdp, clientlen;
25     struct sockaddr_in clientaddr;
26     pthread_t tid;
27   } locals;
28     union { UnStackaccept accept; } children;
29 } UnStackecho;
30
31 Echo server translated to UnStacked C:
32 int echo(UnStackecho * ctx) {
33   switch(ctx->state){ default:
34     ctx->locals.listenfd = open_listenfd(ctx->ops.args.port);
35     while (true) {
36       ctx->locals.clientlen = sizeof(ctx->locals.clientaddr);
37       ctx->locals.connfdp = malloc(sizeof(int));
38       //Calling accept
39       ctx->children.accept.args.sockft = ctx->localslistenfd;
40       ctx->children.accept.args.addr =
41                                   (SA *) &ctx->locals.clientaddr;
42       ctx->children.accept.args.addrlen = &ctx->locals.clientlen;
43       ctx->children.accept.state = INIT;
44       ctx->state = 1;
45   case 1:
46       if (accept(&ctx->children.accept) != 0)
47         return 1;
48       *ctx->locals.connfdp = ctx->children.accept.ops.retval;
49       pthread_create(&ctx->local.tid, NULL, thread, ctx->local.connfdp);
50     }
51     return 0;
52   }
53 }
```

Listing 39: A simple Echo Server implemented using a thread library and then translated to `UnStacked C`

Function Signatures

Every function in the input program is modified in two ways to support stackless continuations. First, the function signature is modified to return an eight-bit signed integer (Line 32 in Listing 39). This new return value is used to signal the *resulting state* of the routine. In particular, the state of a routine is used to properly make

the determination of where that routine continues at a later point in the program's execution. Worth stressing, the state returned here is simply whether or not the routine is blocked. This return value does not capture either the *thread state* or the re-entry point in the function. Second, in addition to the entry point, each function also needs to "remember" where it left off in terms of its context. In order to provide this, we modify the signature of each function to replace its argument list with a single new argument (Line 32 in Listing 39). This new argument is a pointer to a structure that contains all of its context. The context includes the current state of the function. In addition, the context also includes the values of all of its local variables, and the context(s) of any blocking child function call(s).

Context for Blocking Calls

For every blocking function, a structure is generated that can maintain its context (Lines 17–29). The context structure stores the following information about the function while it is blocked:

- *Current state of the function. (Line 18)* This state value is used to determine the entry point of the function upon continuation.
- *Arguments to the function. (Line 20)* Since the function call is no longer guaranteed to be made exactly once (re-entrancy because of blocking child calls), the arguments to each function are maintained in the context structure within the function. This way, regardless of the number of times a method is invoked, the arguments need never enter the system stack.
- *Values of all non-static local variables. (Line 23–27)* Since blocking functions are translated into continuations, the values of local variables must be stored for the duration that the function is blocked waiting on a child blocking call.

- *Contexts of all blocking child function calls. (Line 28)* For each blocking child function call that a parent function makes, the parent maintains a separate context. These child contexts are all part of the parent function's context. In order to optimize the storage space required for these child contexts, we store them in a `union` (since only one blocking child function can be active at any time anyway).

- *Return value. (Line 21)* Since the signature of the function has been modified to return the state of the function, we now store the return value in the context.

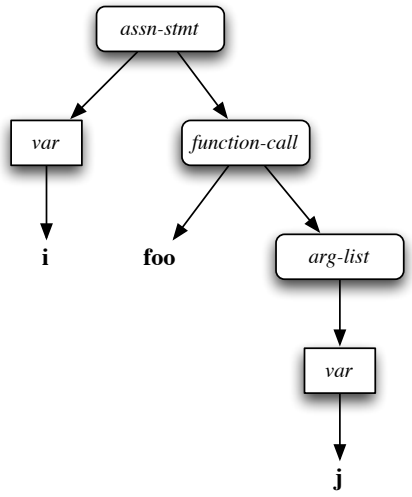


Figure 14: AST for `i = foo(j);` in C

Blocking Function Calls

When translated from multithreaded code to event-driven state machines, blocking function calls have to be transformed into non-blocking (split-phase) calls. In essence the translation "rips the stack" automatically. For example, consider the blocking call to `accept()` on line 10 in the threaded version of `echo()` in Listing 39. This call is translated in `UnStacked C` through a series of steps:

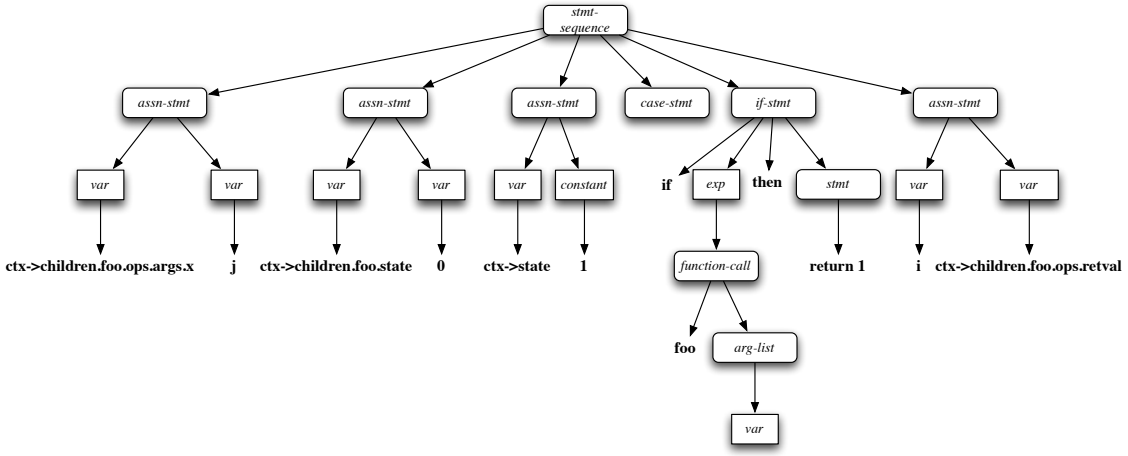


Figure 15: AST for `i = foo(j);` after translation to `UnStacked C`

1. The state of the current function (caller) is changed to mark the current location (Line 44), and a label is placed in the code prior to the blocking call (Line 45). This label will enable execution to resume from this point the next time this function continues.

2. The state of the *child* blocking call is set to `0` (Line 43).

3. The arguments to be sent to the child function are populated in the context structure (Lines 39–42).

4. The call to the child function is made, with a reference to the portion of the context structure that corresponds to this child function as argument (Line 46). If the call to the child function returns any value other than `0` (meaning that the child function is blocked), then the calling function will block as well and return `1` at this point (Line 47). On the other hand, if the child function does not return a blocking state, then the current function will proceed with execution.

Main Routine

In order for any function in the input program to make a call to any blocking function, this calling function must itself be blocking. Therefore, the `main()` routine in the program must not make any blocking calls. If it does the translation converts the `main()` routine into a blocking function. Then a new main with a scheduler inside of it must know to execute the `main()` thread immediately.

Order of Operations

When ripping the stack the way we do, one very important thing to consider is how that affects the order in which operations are executed. This problem manifests itself in several different scenarios. Consider the following assignment statement,

```
1 i = foo(j);
```

This assignment statement has multiple actions in the same statement. First, the function `foo()` needs to be evaluated, and then the return value is assigned to `i`. If `foo()` is a *blocking* operation, there needs to be special attention paid to this line. The blocking operation `foo()` needs to be made in accordance with the translation rule above. Consequently this code is translated as follows:

```
1   ctx->children.foo.ops.args.x = j;
2   ctx->children.foo.ops.state = 0;
3   ctx->ops.state = 1;
4 case 1:
5   if (foo(&ctx->children.foo) != 0)
6     return 1;
7   i = ctx->children.foo.ops.retval;
```

Visualize this in terms of the abstract syntax tree (Figure 14) traversal required for this translation to occur correctly. It is not possible to figure out that `foo()` is a blocking operation at the point of traversing the *assn-stmt* node in the AST. It is only when the traversal has moved down *past* this node, and into the *function-call* node for `foo()` that this determination can be made. At this point therefore, the traversal has to back-track to the assignment statement and *split* this single statement into

a compound statement that looks like the AST presented in Figure 15. In essence, when a blocking operation is one of many expressions occurring in a single statement, that blocking operation needs to be stripped off to *precede* the remaining statements.

There is more. Consider the following statement,

```
1 i = foo(j) + bar(k);
```

where `foo()` is a blocking operation. This code, when translated to `UnStacked C`, becomes the following (just like in the previous example):

```
1   ctx->children.foo.ops.args.x = j;
2   ctx->children.foo.ops.state = 0;
3   ctx->state = 1;
4 case 1:
5   if (foo(&ctx->children.foo) != 0)
6     return 1;
7   i = ctx->children.foo.ops.retval + bar(k);
```

However, what if `foo()` was non-blocking, but `bar()` was blocking? Simply applying the rule of moving `bar()` before the rest of the statement is not going to work: the execution of `foo()` now will come *after* `bar()`, violating the order of operations in the original program. The execution of `foo()` In this case, therefore, the correct translation should be as follows:

```
1   ctx->children.foo.ops.retval = foo(j);
2   ctx->locals.tmp = ctx->children.foo.ops.retval;
3   ctx->children.bar.ops.retval = k;
4   ctx->children.bar.ops.state = 0;
5   ctx->state = 1;
6 case 1:
7   if (bar(&ctx->children.bar) != 0)
8     return 1;
9   i = ctx->locals.tmp +
10        ctx->children.bar.ops.retval;
```

Variables

A key feature of `UnStacked C` is that the stack is not used for maintaining local variables in blocking operations. Instead these variables are maintained in the contexts of these blocking operations. Global variables are not touched in this translation. Similarly with static variables. The AST of the program is traversed to transfer the

local variables from blocking functions into context structures that correspond to each blocking function.

While the declarations of local variables are *transferred* to context structures, the *initialization* statements are retained in the same place, and only *names* of the variables are changed:

```
1 int i = 7;
```

is translated to:

```
1 ctx->locals.i = 7;
```

Here again, order of operations matters. If the right hand side of the assignment is a blocking function call, then that blocking function is moved above the assignment.

5.2 Implementation

`UnStacked C` is implemented as a C-XML-C transform. It is written in Python and performs its transformation in two passes of the AST. During the first pass it analyzes the AST. The second pass performs the transformation.

The first pass gathers information about all of the functions in the system. One core piece of information gathered is whether a given function is marked as blocking. Another piece of information gathered is the call graph. Also all local variables and arguments are gathered. After the first pass completes, we can analyze the call graph and the blocking functions to determine which functions in the system must block (since they call another blocking function). We can then generate all of the possible blocking context data types.

The second pass inserts all of these blocking context data types prior to the first modified function, as a way to ensure that they are declared prior to usage. Next it transforms each blocking function call's declaration to match the new function

signature. Finally it transforms each blocking function call's function body in the order that they originally appeared.

5.2.1 Modes of Operation

There are several different configurable portions of `UnStacked C`. These configuration options allow developers to integrate `UnStacked C` into many different environments. They also allow developers to attempt to gain different levels of performance by using different options at compile time, without needing to change the source code. It is important to note that currently, all of these modes require the complete source code of the application.

There are three main modes of different `UnStacked C` function body transforms.

1. The first one was shown in Listing 39. It uses Duff's device but precludes the use of switches in the rest of the system. This means that if a user has a simple switch case, CIL may transform it into a series of if statements. If the switch statement is more complicated then CIL will leave it in place, so that the final compiler can make a jump table. If CIL leaves the switch statement in place in a blocking function, then the transform will fail. This led to using different transforms.

2. The second type of function body transform uses a GCC extension known as *label as values* [68]. This extension allows one to store a label in a variable, and they branch to it later on. We translate the same function initially shown in Listing 39 with the labels as values and show the results in Listing 40. When using Labels as Values, we have only made five changes compared to the Duff's device transformation. The first change is to the type of the state variable, now it is a *void **. It is show on line 3 of Listing 40. Line 28 shows the child's state

being reset to *NULL* prior to execution. Line 18 shows a distict difference where a single goto is used to branch to the label stored if it is not NULL.

Line 29 shows the label value being stored into the state using the `&&` operator. The last change can be found on line 30 and it contains the new label.

```
1  Data structure to store context for continuation:
2  typedef struct UnStackecho {
3    void * state;
4    struct {
5      struct { int port; } args;
6      int retval;
7    } ops;
8    struct {
9      int listedfd, *connfdp, clientlen;
10     struct sockaddr_in clientaddr;
11     pthread_t tid;
12   } locals;
13     union { UnStackaccept accept; } children;
14 } UnStackecho;
15
16 Echo server translated to UnStacked C using Label as a Value:
17 int echo(UnStackecho * ctx) {
18   if(ctx->state)goto *(*ctx->state);
19   ctx->listenfd = open_listenfd(ctx->ops.args.port);
20   while (true) {
21     ctx->locals.clientlen = sizeof(ctx->locals.clientaddr);
22     ctx->locals.connfdp = malloc(sizeof(int));
23     //Calling accept
24     ctx->children.accept.args.sockft = ctx->localslistenfd;
25     ctx->children.accept.args.addr =
26                                  (SA *) &ctx->locals.clientaddr;
27     ctx->children.accept.args.addrlen = &ctx->locals.clientlen;
28     ctx->children.accept.state = NULL;
29     ctx->state = &&UnstackedC_echo_1;
30   UnstackedC_echo_1:
31     if (accept(&ctx->children.accept) != 0)
32       return 1;
33     *connfdp = ctx->children.accept.ops.retval;
34     pthread_create(&ctx->tid, NULL, thread, ctx->connfdp);
35   }
36   return 0;
37 }
```

Listing 40: The Echo Server translated using Labels as Values

3. The last mode is in between the other two translation modes: generating a branch table to jump to specific labels as shown in Listing 41. Like Duff's device, it only needs to store a single byte for state, and like the Labels as Values it uses labels throughout the source code. The branch table can be found on line 18 of Listing 41. Lines 29 and 30 show the new state value and labels respectively.

```
1  Data structure to store context for continuation:
2  typedef struct UnStackecho {
3    void * state;
4    struct {
5      struct { int port; } args;
6      int retval;
7    } ops;
8    struct {
9      int listedfd, *connfdp, clientlen;
10     struct sockaddr_in clientaddr;
11     pthread_t tid;
12   } locals;
13     union { UnStackaccept accept; } children;
14 } UnStackecho;
15
16 Echo server translated to UnStacked C using custom branch table:
17 int echo(UnStackecho * ctx) {
18   if(ctx->state == 1)goto UnstackedC_echo_1;
19   ctx->listenfd = open_listenfd(ctx->ops.args.port);
20   while (true) {
21     ctx->locals.clientlen = sizeof(ctx->locals.clientaddr);
22     ctx->locals.connfdp = malloc(sizeof(int));
23     //Calling accept
24     ctx->children.accept.args.sockft = ctx->localslistenfd;
25     ctx->children.accept.args.addr =
26                                 (SA *) &ctx->locals.clientaddr;
27     ctx->children.accept.args.addrlen = &ctx->locals.clientlen;
28     ctx->children.accept.state = 0;
29     ctx->state = 1;
30   UnstackedC_echo_1:
31     if (accept(&ctx->children.accept) != 0)
32       return 1;
33     *connfdp = ctx->children.accept.ops.retval;
34     pthread_create(&ctx->tid, NULL, thread, ctx->connfdp);
35   }
36   return 0;
37 }
```

Listing 41: The Echo Server translated using a generated branch table

Since developers have a choice we ran several tests to determine the switching time. We calculated the individual context switching time of a thread while varying the number of yields in a function. We measured the time it takes for 1000 iterations of a loop to complete. Dividing the measured time by 1000 gives us a good approximation of the context switching costs. It will also take into account any scheduling costs, but those will be constant across the different techniques.

These results were measured on a 16bit Microchip PIC24FJ[1] and a 64bit AMD Athlon.[2] Other processors or optimization levels will likely yield different results.

[1]The program was compiled for the PIC24FJ128GA010 using Version 3.24 of Microchip's C30 compiler with the -Os optimizations turned on. It was run using the MPLAB SIM30 version 4.30 at 4MHz.

[2]The program was compiled using GCC 4.4.5 with the -O3 optimizations on an AMD Athlon^{TM}64 X2 Dual Coe Processor 3600+

These results provides some guidelines to users if more optimizations are desired.

First we consider the PIC24FJ results. Figure 16 shows the context switching time and Figure 17 shows the additional program size for each additional yield point. Not shown in these graphs is the additional 3 bytes of RAM each context frame of the Labels as Values method uses. It appears that all three methods end up optimizing to the same size and speed if there is only one yield point. The second yield point shows a distinct difference where the Labels as Values method appears to use significantly larger amounts of code space. This is because it must load the full address into the state and then branch to it. Notice how the amount of time goes down significantly as the number of yields in a given function increases. This is due to the fact that any initial overhead it incurs in indirectly branching to an address is overcome eventually when enough yield points exist in a single function.

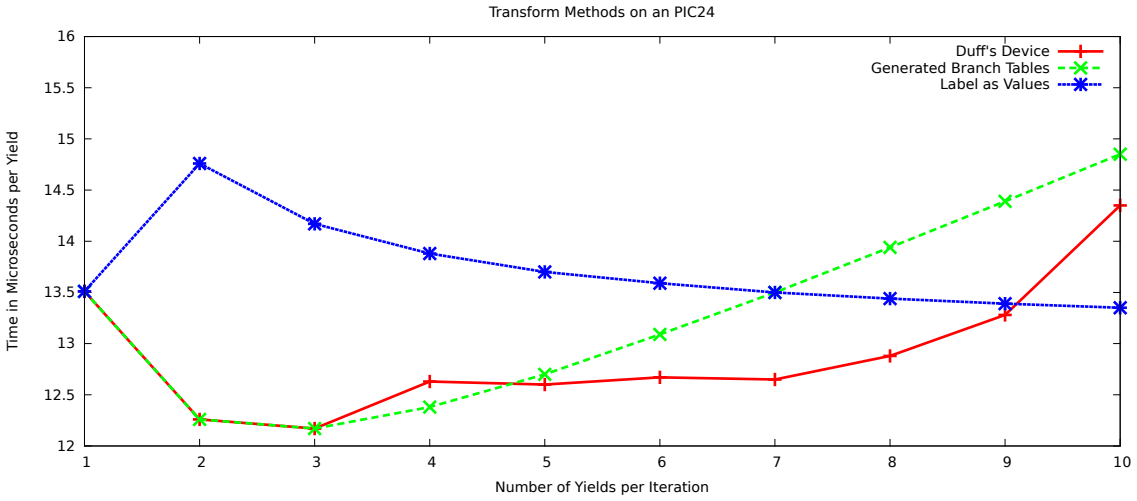


Figure 16: Context switching times in a PIC24 of different transformation methods.

For functions which have many yield points, Labels as Values appears to be the clear winner, with Duff's device and the generated branch tables as a second. As far as program space goes, Duff's device appears to vary the most, since it isn't a simple linear addition of programming space but it appears that the compiler is switching between jump table implementations.

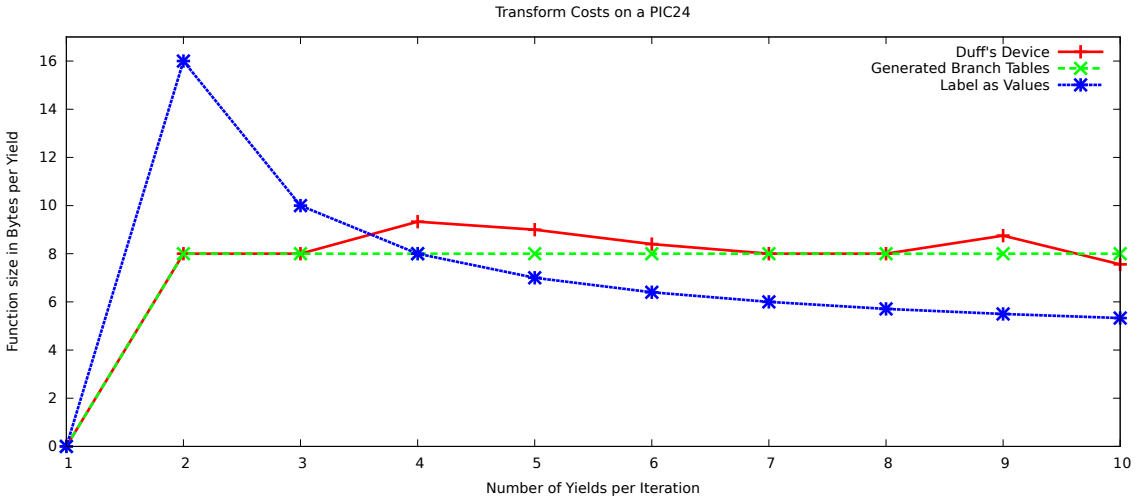


Figure 17: Program size of different transformation methods in a PIC24.

The results for the AMD CPU give a very different set of results. Figure 18 shows the execution performance of the different methods. In this case, the generated branch tables give the fastest results, with Duff's device appearing to decrease performance when the number of yields is five or greater. The costs for the different transform methods on the AMD is shown in Figure 19. On the AMD CPU, the cost of the different methods is fairly similar, with Duff's device appearing to secure the smallest overheads of any of the methods.

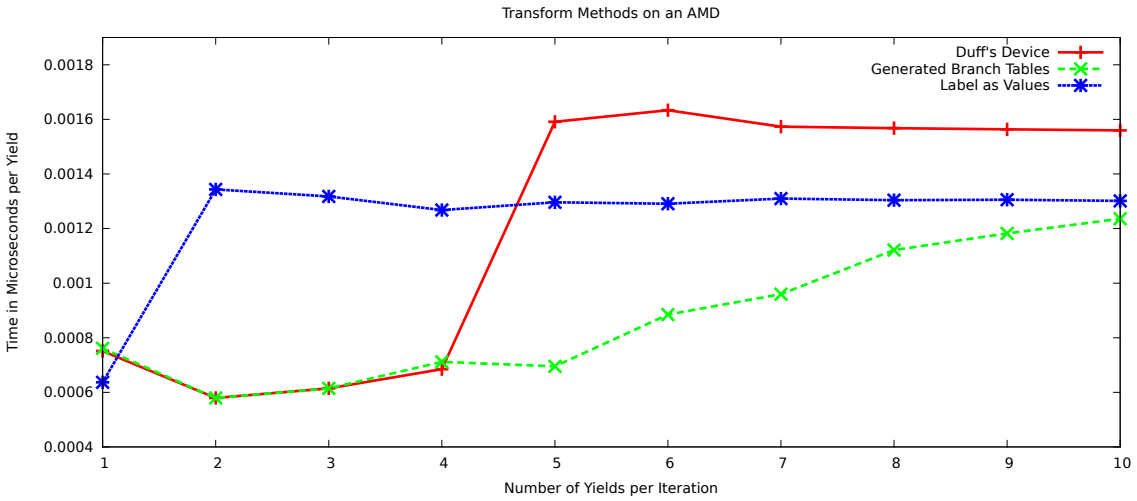


Figure 18: Context switching times in an AMD CPU of different transformation methods.

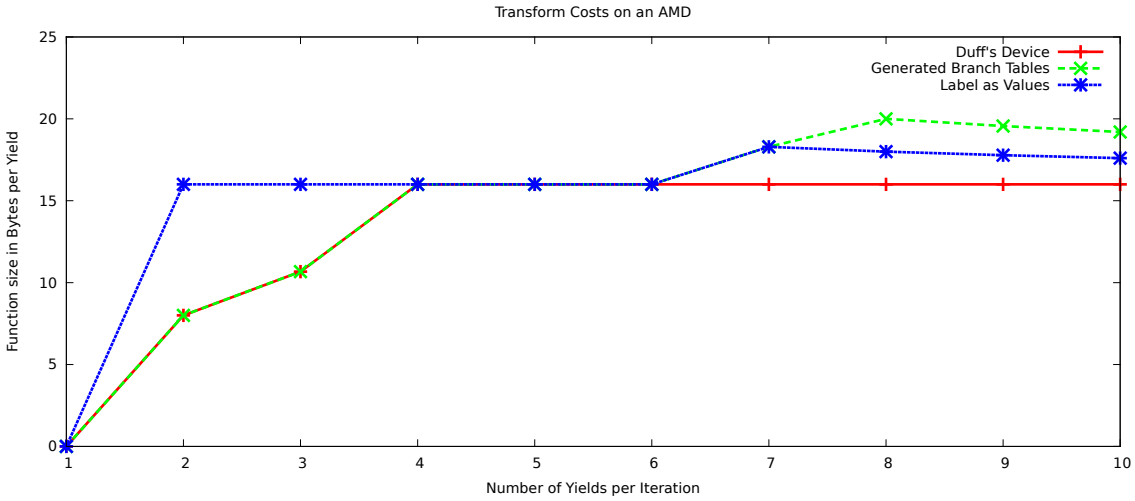


Figure 19: Program size of different transformation methods in an AMD CPU.

Currently `UnStacked C` defaults to using generated jump tables since it allows developers to use any C compiler they want and still be able to use switch statements. Since the values in Figures 16 and 17 are based upon one compiler and one hardware platform, it is entirely possible that other hardware platforms will perform differently. The transform method is selectable on the command line and in the C-XML-C config.ini file.

5.2.2 C Extensions

Every threading framework has a different set of low level primitives. `UnStacked C` can work inside of all of these frameworks by adding a handful of hooks to the language through attributes. This allows `UnStacked C` to understand the already existing names. Attributes in C are a common way of extending capability without breaking compatibility. We introduce two new attributes which we use to support stackless threading in C. These two attributes are *yield* and *blockingcall*.

The first attribute we added is named *yield*. It is used to mark a function which causes the routines to yield to the scheduler, or to the next thread. Some operating systems call it something like `yield` or `yieldTask`. Others call it

`suspendCurrentThread` or something similar. One restriction of `UnStacked C` is that whatever the yield function is, it must take no arguments and return no values. We have found no cases where an unconditional yield needs any arguments nor any return values.

The other attribute is *blockingcall*. It is used to allocate the exact amount of stack space for a given thread. Without `UnStacked C` the stack size is an educated guess at the amount of stack space needed. A variable marked with *blockingcall* notifies the compiler that it needs to change the type of a given variable to match a function's context. *blockingcall* takes an argument of the name of the function of the desired stack space. It can take multiple arguments which means it will allocate a union of those possible stacks, which will allocate the space required for the largest of those threads. The context (what used to be stack space) can be allocated in the following way:

```
1 int i __attribute__ (blockingcall(fun_name));
```

We also implemented a macro that can be used in place of the `blockingcall` attribute:

```
1 UNSTACKED(fun_name) i;
```

These allow developers to only allocate the context space they need, with no risk of overflowing the thread stacks.

The `UnStacked C` Scheduler

Every other threading framework has some sort of a scheduler except for `UnStacked C`. Instead of having its own scheduler, it can operate with whatever scheduler the underlying RTOS uses. This allows `UnStacked C` to be injected into many different kinds of systems, without the need to customize `UnStacked C`.

5.3 Evaluation and Results

In order to evaluate the performance of `UnStacked C` threads, we designed a simple (somewhat contrived) application. The application is intended purely as a proof-of-concept to demonstrate `UnStacked C`'s potential. This program is not an application design toward embedded systems, instead it demonstrates the scalability of `UnStacked C`. The program sends a token around a ring of one million threads. When a thread receives the token, it simply passes it on to its neighbor.

Experiment Setup. We configured this program to run with increasing numbers of threads, and measured three metrics: thread creation time, overall execution time (including context-switching time), and the memory usage. The experiments were run on a laptop running Linux, with 2 GB RAM. In the application, all the threads are allocated and created statically. We compiled the program with regular cooperative threads, and then the same program translated using our compiler to `UnStacked C`. Note that we could not proceed beyond 100,000 threads (on log scale) with regular cooperative threads, whereas we could get up to 10 million threads for the same program with `UnStacked C` before the computer ran out of memory.

Thread Creation Time. The first experiment compares how long it takes for all the threads in the program to get created. Figure 20 shows this comparison. The thread creation time grows linearly with the number of threads in both cases. For the cooperative thread case, the thread creation time grows from 6.3×10^{-5} seconds for 10 threads to 2.94×10^{-1} seconds for 100,000 threads. By comparison, the thread creation time for `UnStacked C` grows from 1.89×10^{-5} for 10 threads to 7.71×10^{-3} for 100,000 threads to 7.71×10^{-1} for 10 million threads. This linear growth is consistent with expectation: each thread needs to be initialized sequentially. These results are not a function of memory allocation as all of the stacks are pre-allocated

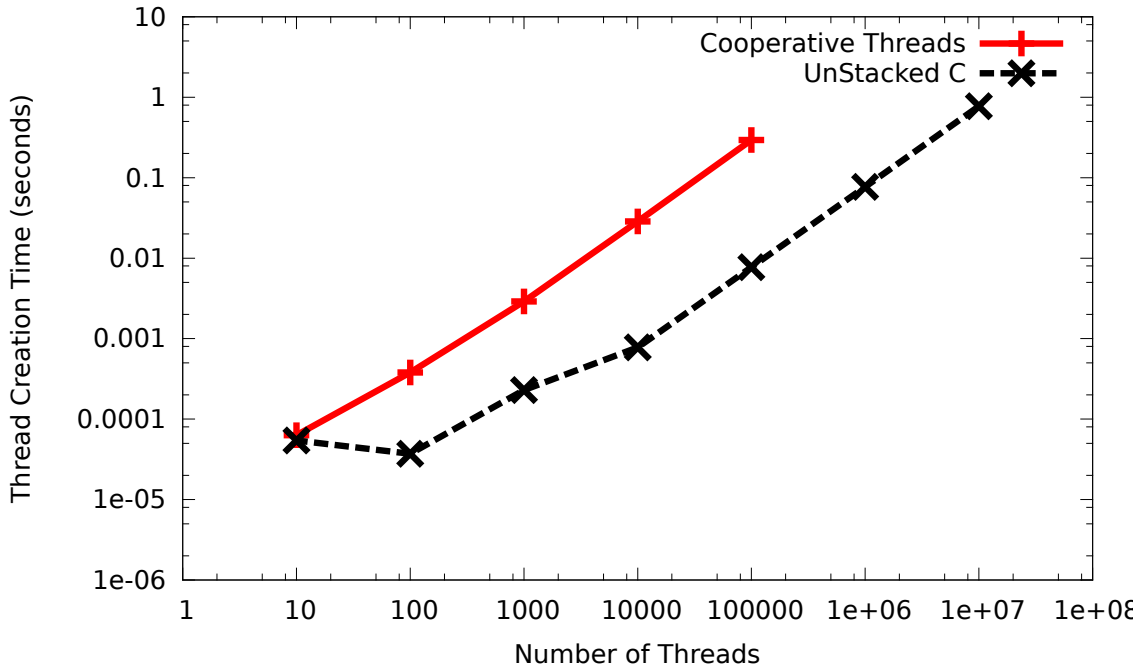


Figure 20: Comparing thread creation time of cooperative threads vs. `UnStacked C`. (Both axes are presented in log scale.)

with the memory initialized, so that no allocation occurs at runtime. As can be observed from the figure, `UnStacked C` consistently outperforms the cooperatively threaded implementation.

Execution Time. The second experiment compares the execution time required for sending 100 token messages around the ring. Figure 21 shows this comparison. In the case of the cooperative thread implementation, the time taken grows linearly from 1.01×10^{-3} seconds for 10 threads to 11.35 seconds for 100,000 threads. In the case of the `UnStacked C` implementation, the execution time grows (linearly, again) from 4.5×10^{-5} seconds for 10 threads to 8.63×10^{-1} seconds for 100,000 threads to 86.1 seconds for 10 million threads. Again, the performance of `UnStacked C` is consistently better than the cooperative thread implementation (by two orders of magnitude).

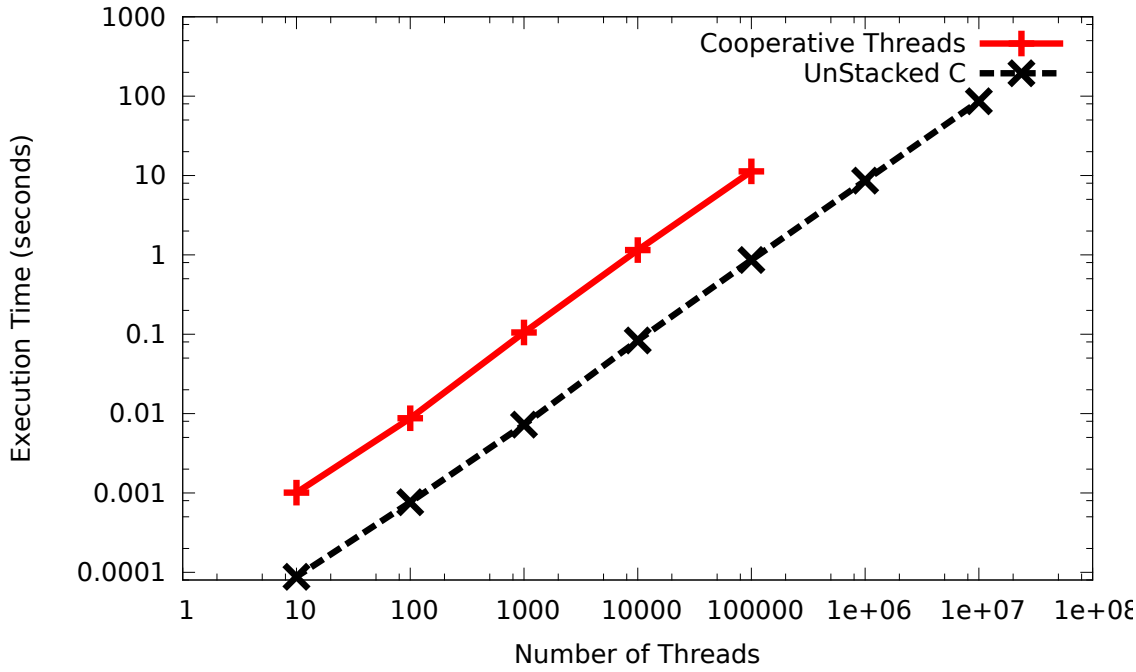


Figure 21: Comparing execution time of cooperative threads vs. `UnStacked C`. (Both axes are presented in log scale.)

Memory Usage. Finally, we measured the amount of memory used by the token ring application. This is probably the most important measure, given that the available memory is what limits the number of threads. To make as "close" a comparison between the two implementation, we set the stack size for the cooperative thread implementation to as small as possible (2 KB).[3] For the cooperatively threaded implementation, the memory use grows from 1.6 MB for 10 threads to about 238 MB for 100,000 threads. The `UnStacked C` implementation grows from 1.58 MB to about 6.6 MB for 100,000 threads to 509 MB for 10 million threads. As can be seen from the comparison graph, the memory use grows at a *significantly* slower rate than the cooperative threaded implementation. This is a manifestation of the fact that `UnStacked C` does not use fixed-size stacks for each thread.

[3]At this level, the stack is already too small. While the program executes fine during normal operation, any UNIX Signals cause the stack to overflow.

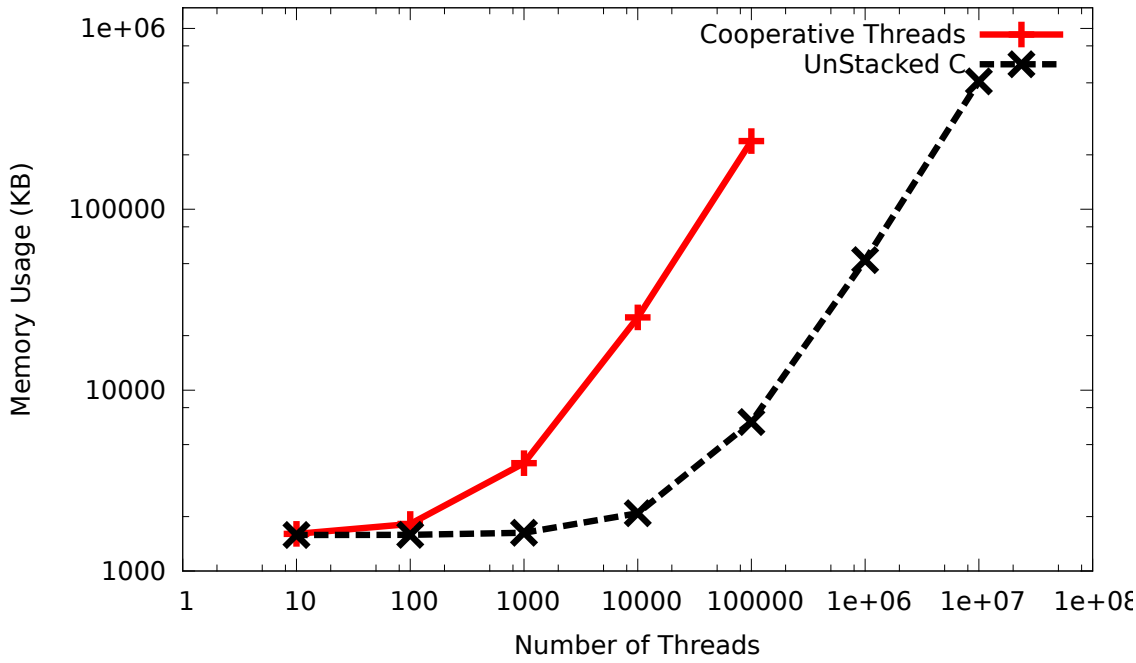


Figure 22: Comparing memory usage of cooperative threads vs. `UnStacked C`. (Both axes are presented in log scale.)

5.3.1 TinyThread Comparison

To further evaluate `UnStacked C` we added support for it inside of TinyThread [50]. TinyThread is a threading library for TinyOS which runs on a variety of different hardware platforms. TinyThread uses our *stack-estimator* to calculate the exact amount of stack each thread requires. This allows TinyThread to have the smallest RAM overhead of any previous embedded multithreading system, since only the theoretically worst case RAM is allocated. This comparison gives a distinction on how the RAM usage can be reduced beyond previous theoretical limits by using `UnStacked C`.

We had to modify a single line of the source code in the applications to support `UnStacked C`. Listing 42 shows the change made to the source code. This line allocates the stack space for a given thread. We changed it to add the *blockingstack* attribute, notifying `UnStacked C` to change the type of this object to match the

context of the function. This change is very minor, and only needs to be done once per function. This change could have been implemented as a C-XML-C transform, but it would have been completely specific to TinyThread.

```
1 //Original Thread Stack allocation (threadBlink1_STACKSIZE is declared by the stack
      analyzer)
2  uint8_t blink1Stack[threadBlink1_STACKSIZE];
3 //Modified code for UnstackedC, telling the compiler to change the type for threadBlink1s
      context
4 int __attribute__((blockingstack("threadBlink1")))  blink1Stack[1];
```

Listing 42: Changes made to TinyThread's stack allocation

We ran our evaluation on the five applications included in the TinyThread distribution. We made the aforementioned modifications and recompiled all of them. We tested them all on T-Mote Sky motes, which contains a MSP430F1611 microcontroller [53] and they all operated properly. Figure 23 shows the memory (RAM) comparison, while Figure 24 shows the program memory (ROM) comparison.

`UnStacked C` reduces the RAM consumption by an average 27.8% versus the theoretically smallest stack size possible. The rest of the RAM consumption in the system remains constant, only the thread stack consumption changes. So the reduction is essentially removing the RAM overhead caused by multithreading.

`UnStacked C` does come at a cost. It increased the ROM by an average of 6.9%. We deem this cost is acceptable. Microcontrollers have many times more ROM than RAM. Also these numbers are statically allocated at compile time, letting a programmer know exactly how much RAM and ROM a given program actually needs.

Table III shows a list of the number of threads in each of these programs. Notice that it is not just the number of threads the influences the RAM usage, but it is also what the programs are doing. For instance, `Blink`, `BlinkBarrier` and `SenseT` do not use the radio. This leads tp a significantly lower interrupt overhead on each thread because many interrupts go unused. Applications which use the radio such as `Bounce` and `SenseBroadcastT` require significantly more RAM even when

they only have one application thread running.

Program Name	Number of Threads
Blink	3
BlinkBarrier	2
Bounce	1
SenseT	1
SenseBroadcastT	1

Table III: The number of threads in each program.

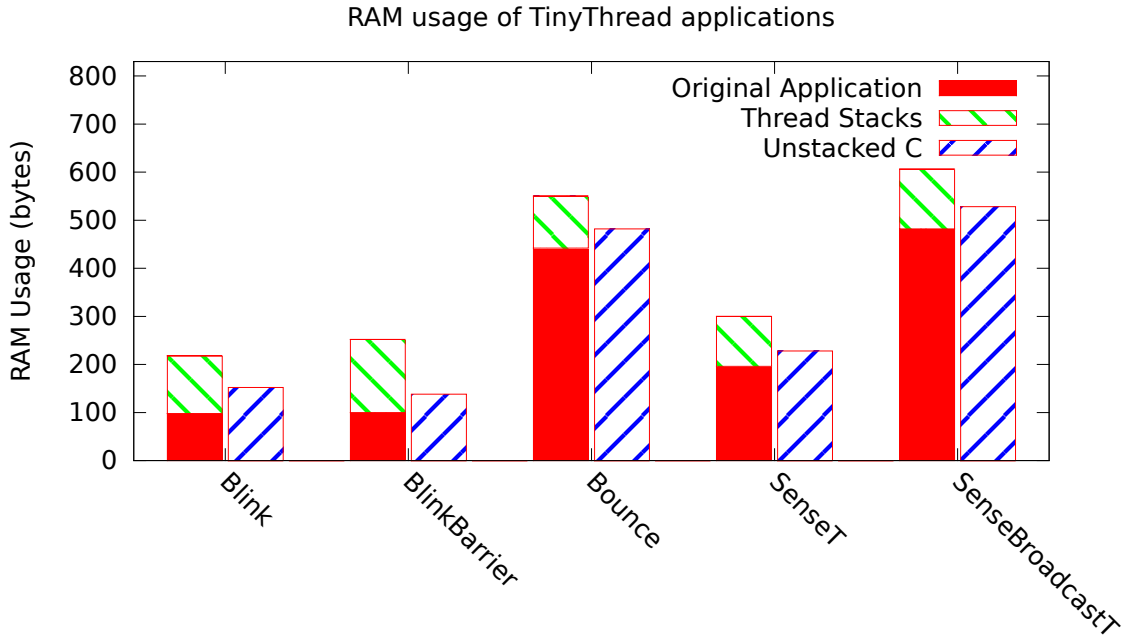


Figure 23: Comparing memory (RAM) usage of TinyThread vs. TinyThread with `UnStacked C`.

5.3.2 Limitations

Our current implementation of the `UnStacked C` has a few limitations.

1. The current version of `UnStacked C` does not support recursion. A different

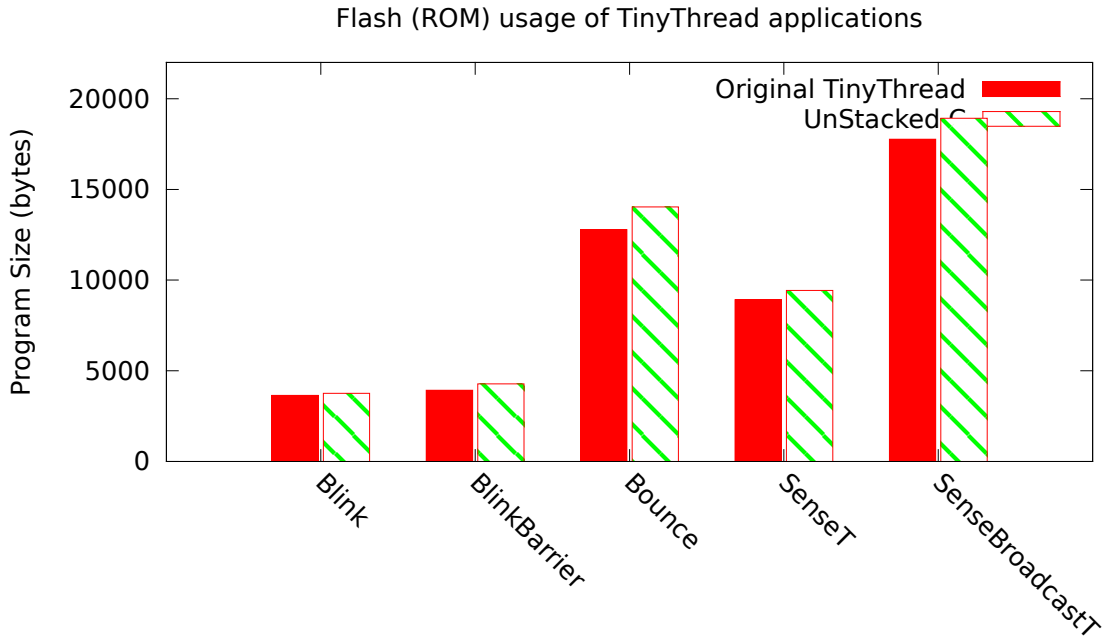


Figure 24: Comparing program memory (ROM) usage of TinyThread vs. TinyThread with `UnStacked C`.

implementation of `UnStacked C` can be used to address recursion.[4] If recursion is allowed the precisely required stack size cannot be calculated at compile time, and developers may accidentally use recursion. Instead of implementing recursion we decided it was better to reduce faults due to stack overflows and force developers to avoid recursion.

2. We also do not support indirect function calls of blocking functions. This means that blocking functions cannot be invoked through function pointers. This limitation can be removed if the Indirection Removal transformation from Chapter 4 is applied first.

3. All blocking calls must have a fixed number of arguments. Functions with

[4]To change the transform to support recursion, we need to change the way contexts are defined. We need to break the context into two parts. One part with the arguments, state and return values; Another part with the local variables and child calls. This way the caller only needs to see the first part of the context, and the second part of the context need not be allocated. This creates a system with essentially a dual stack and new possibility for stack overflows in the context stack.

variable numbers of arguments can exist in the code, but they cannot block.[5]

4. There is no incremental compilation. When a single part of a program is modified, and needs to be re-compiled, *everything* in the entire application needs to be re-compiled. In other words, pre-compiled libraries cannot be used, at least not in a blocking context. Their source code could be added to the project, and they too would be in-turn compiled into the application. This is an artifact of all whole-program compilers [78] and not specific to `UnStacked C`.

5.4 Summary

In this chapter, we have presented `UnStacked C`, a C-to-C translation approach to building stackless C continuations. The most important contribution of the work presented here is that it enables richer design strategies that were previously too "cost-prohibitive" in terms of memory utilization. It grants these design strategies without inherent risk of stack overflows.

The `UnStacked C` translator that we have implemented takes as input a C program written using cooperative threads and automatically generates the corresponding program that is an event-driven state machine. It performs this with only minor, if any, changes of the applications themselves.

Our simple proof-of-concept application can run with *a million threads* while occupying less than 50 MB of RAM. We see this as a paradigm leap: design strategies that were previously completely ignored can now be re-visited, and actually implemented.

More importantly, for embedded system programming, we see `UnStacked C` as an enabler to designing richer functionality on low-resource devices. Sensor nodes,

[5]This is a symptom of the implementation, and could be fixed by a more complicated transformation of the source code. It is also possible to fix this limitation with another transform that removes the variable number of arguments.

for example, that have been designed to be dumb data collectors because of the expressiveness limitations of the programming model can now be armed with extra functionality. We have applied `UnStacked C` to a number of existing embedded applications and have reduced memory consumption by more than 25%.

The current implementation of the `UnStacked C` compiler is a C-to-C translator built in Python that uses C-XML-C to parse cooperatively multithreaded C programs, and then generates the corresponding event-driven program. In Chapter 6 we extend `UnStacked C` to support preemptive threading with Lazy Preemption.

CHAPTER VI

UnStacked C with LP

Lazy Preemption is a method of translating preemptive multithreads into cooperative multithreads. We implement `UnStacked C` with Lazy Preemption inside of `UnStacked C` as a C-XML-C transform and call it `UnStacked C with LP`. We will evaluate it against TOSThreads, an embedded preemptive multithreading system.

Preemptive multitasking describes a computer system where the tasks can be preempted, or seized from the processor, stored, and then the next task is allowed to run for some time. This allows many tasks to run seemingly simultaneously on a computer system. It also means that each task gets a certain amount of time on the processor prior to preemption. In a cooperative multitasking system each task runs until it voluntarily releases the processor to the next task. In a cooperative multithreading system there is no basis in time, instead it is when a task yields to the next task.

Preemptive multitasking is fairly safe since there is little or no interactions

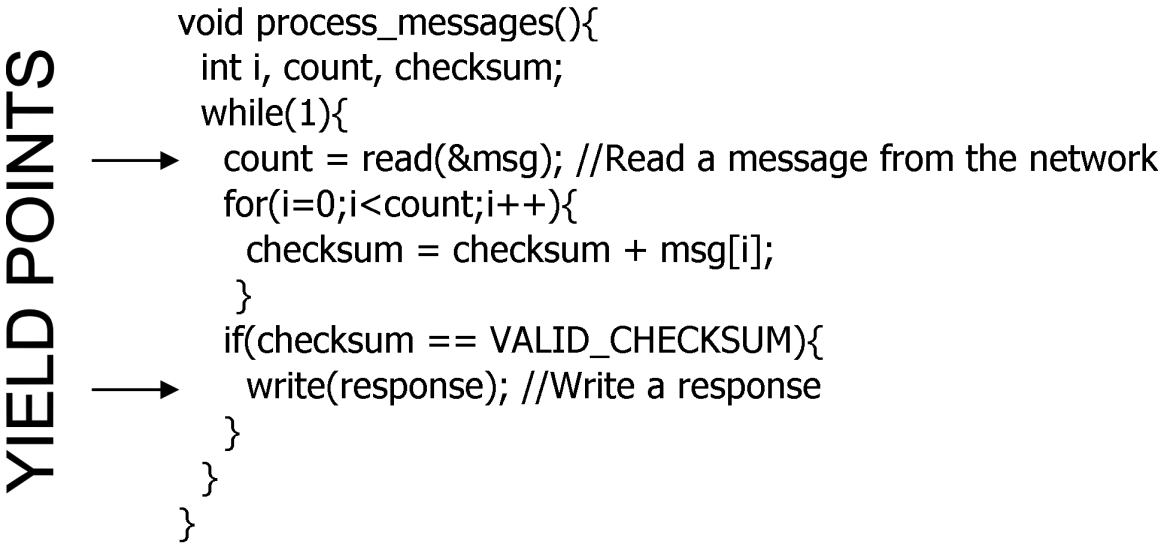


Figure 25: Yield points for an example function with cooperative multithreading.

between the separate tasks. In the case of preemptive multithreading[1] the risks become significant since the threads share memory. The issues occur because multiple threads can be attempting to use the same resource, in this case memory, at the same time. Many of these issues do not occur with cooperative threads.

Cooperative multithreading has a different set of problems. Cooperative multithreading relies on each thread to relinquish control. What happens if a thread does not relinquish control in a timely manner? Then the entire system can be slowed or even hung because of a single thread.

A cooperative thread can only yield at specifically defined points. This allows programmers to reason about how the system will operate. In fact they can reason about it as a finite state machine. An example program using cooperative threads is shown in Figure 25, with the possible yield points marked. These yield points are easy for programmers to identify.

In preemptive multithreading, programmers cannot identify all of the possible yield points from the source code. Preemptive multithreading allows preemption to

[1]Tasks are analogous to processes or programs on a PC. Threads are a type of task which all share memory with each other.

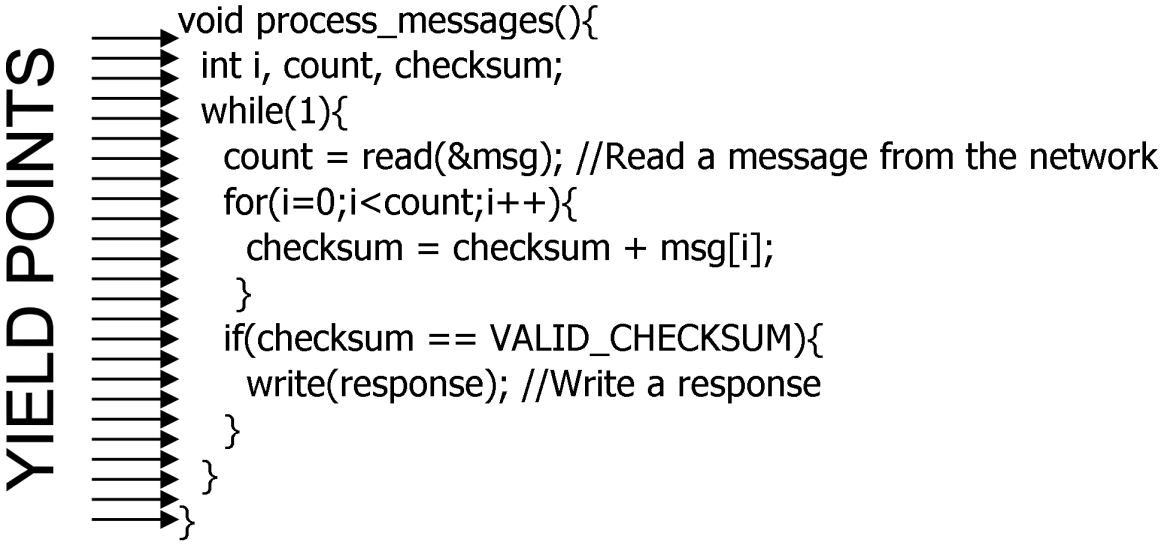


Figure 26: Yield points for an example function with preemptive multithreading.

occur at any instruction. A single line of source code may translate into zero to many instructions. Since a programmer cannot know exactly how many instructions a given line of source code will emit, they cannot know all of the different ways it can be preempted. In Figure 26, all of the preemption points cannot be identified by programmers. After a thread is compiled, it does have a limited number of instructions, so the number of yield points becomes finite. Prior to compilation, developers must assume that there is an infinite number of preemption points in any given preemptive thread.

These problems can be distilled down to preemptive multithreading having too many yield points and cooperative multithreading having too few. It would make sense that some middle ground can be found, namely Lazy Preemption. Figure 27 shows the yields points to the same function if Lazy Preemption is used. When comparing with the cooperative multithreading yield points from Figure 25 notice there are two new yield points added at the end of each loop. These are not typical yield points, instead they conditionally yield based upon the value of some global flag. The flag is set when the scheduler wants to preempt the thread. When the thread gets to a conditional yield point, it reads the value, and if it is set it yields. Holenderski

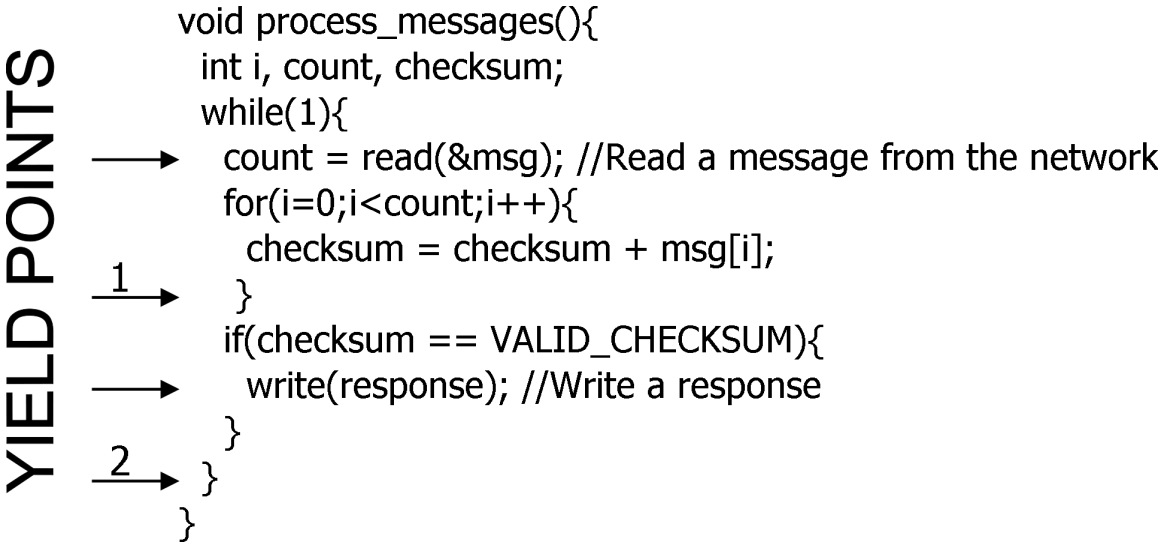


Figure 27: Yield points for an example function with lazy preemption.

et al. [36] calls these *optional preemption* points when they were manually inserting them in the application. This is different than Lazy Preemption, since they chose where these *optional preemption* points were placed, as opposed to putting them in every loop.

6.1 The Algorithm

We implement Lazy Preemption as an option inside of the `UnStacked C` transform. The algorithm can be described in 3 steps:

1. Determine the call graph
2. Identify which functions are blocking and therefore need to be transformed
3. In each blocking function, insert an optional preemption point at the end of each loop, and prior to each goto statement which branches backwards.

We mentioned in the algorithm that we need to know which functions are blocking and we can use the yield statements, but that will not get computational

functions marked as blocking. This is because a straight computational function may not call yield nor any other blocking function, so it will not get transformed by `UnStacked C`. To support this, we added a *blocking* attribute to functions so that developers can mark functions as blocking. This allows them to become translated by Lazy Preemption.

6.2 Analysis

Lazy Preemption injects these optional preemption points, which allows it to preempt only at specific times. Since preemption can only occur at rational points in the source code, it means that all lines of code with non-blocking operations in them are implicitly atomic. This atomicity removes all possibilities of multi-word data faults of a single variable. This stems from a single line of C code being non-preemptable with Lazy Preemption so any multi-word writes can complete.

Since `UnStacked C` forces all writes to complete when it yields (since it is truly a function return). This means that it is not possible to have a temporal data fault between two `UnStacked C` threads. So following that observation, using `UnStacked C with LP` also cannot have temporal data faults between threads. It is important to note that some temporal and multi-word data faults can occur between interrupts and `UnStacked C with LP` threads still, but not between the different threads.

Like Lazy Preemption, Python only allows preemptions to occur at specific points. Python uses a global interpreter lock (GIL) [72] to ensure that only one thread at a time can be executing in its virtual machine (VM).[2] Python's GIL is essentially a mutex which the executing thread holds until a byte-code instruction. This makes every python instruction atomic, even if the operations take many physical

[2] A virtual machine is a software device that acts like its own kind of computer. Instead of running on machine language instructions like a microprocessor it uses an arbitrary instruction set called byte-code.

instructions to execute. This eliminates all temporal and multi-word data faults of a single variable.

6.3 Evaluation and Results

We evaluated Lazy Preemption in two ways. First we measured the overhead Lazy Preemption incurs by manually performing the transform and directly measuring the overhead compared with a non-threaded version. Second, we compare it with TOSThread, which is a preemptive multithreading library for embedded systems, by recompiling the same applications using `UnStacked C with LP`.

6.3.1 Lazy Preemption Overhead

To evaluate Lazy Preemption preemption we use a microbenchmark. The microbenchmark calculates the lowest number and is shown in Listing 28. It has two loops which iterates through all of the numbers until it finds one that evenly divides between the first ten numbers.[3] We added two optional preemption points to the end of each loop as shown in Listing 29. We then ran both applications on a Microchip PIC24FJ256DA210.

Table IV contains the results from running the code in Listing 29 and Listing 28. There is a distinct growth in code size. The growth in code size is a cost and must be understood when using Lazy Preemption. The performance decrease in execution time is negligible. We do not consider this a significant cost.

These costs can be offset in space and performance because other primitives do not have to be employed to protect shared variables. This is doubly true with the performance decrease.

[3]This is taken from problem 5 of Project Euler. `http://projecteuler.net/index.php?section=problems&id=5`

```
1  long int lowerCommonDen1to10(){
2    long int i;
3    int j;
4    for(i=20;i<MAX_COUNT;i++){
5      for(j=2;j<MAX_LEVEL;j++){
6        if(i % j != 0)break;
7      }
8      if(j == MAX_LEVEL)return i; //Return the value
9    }
10   return 0;
11 }
```

Figure 28: Calculation of lowest common denominator of the numbers from one to ten

```
1  long int lazyPreemption(){
2    long int i;
3    int j;
4    for(i=MAX_LEVEL;i<MAX_COUNT;i++){
5      for(j=2;j<MAX_LEVEL;j++){
6        if(i % j != 0)break;
7        if(flag)return 1; //Optional Preemption point
8      }
9      if(j == MAX_LEVEL)return i;
10     if(flag)return 1; //Optional Preemption point
11   }
12   return 0;
13 }
```

Figure 29: Calculation of lowest common denominator of the numbers from one to ten with Lazy Preemption

6.3.2 TOSThreads Comparison

To evaluate `UnStacked C with LP` we modified portions of the TOSThreads kernel to support `UnStacked C`. The first change was to remove the platform specific stack-swapping operations, and replace them with a yield function to notify `UnStacked C`. Next we replaced the thread context switches in the TOSThread scheduler with executions of the `UnStacked C` threads.

We then changed the stack allocation routines to context allocation routines. This is different than TinyThread comparison from Chapter 5, in that the applications in TOSThreads attach the desired stack size and the TOSThread kernel allocates the stack space. We changed the allocator to ignore the user's size and instead allocate the required context size instead. This means that the TOSThread applications do not have to change to be recompiled with `UnStacked C with LP`.

	Calculation	Calculation with Lazy Preemption	Percent Difference
Function Size	66	80	21%
Time	134.9ms	136.0ms	0.82%

Table IV: Calculation time and size with and without Lazy Preemption

The last change involved changing how preemption works with interrupts in TOSThreads with `UnStacked C`. TOSThreads has a unique priority method where it attempts to force threads to be a lower priority than the event-driven system, and it uses a postamble on each interrupt to accomplish it. Normally the postamble must be added to each interrupt handler that checks to see whether it is in a thread and if so, it checks to see if there are any tasks in need of running. If both of these are true, then the thread goes and is preempted in the interrupt handler.

This work also uncovers a bug in TOSThreads where they attempt to preempt (accidentally) inside of an event handler. This is likely due to them attempting to use a software timer to preempt the threading system. Since software timers all share a hardware timer and do not run in interrupt handlers, they cannot actually preempt threads in the system. This serendipitous mistake still allows the system to operate because the software timer posts a task to the main loop, which has a higher priority, which will in turn force a preemption of the current thread. So preemption still works, but there is significant overhead in its implementation.

In `UnStacked C with LP` the postamble simply checks to see if there are more tasks waiting to be executed (which should be treated as higher priority according the Klues et al. [44]) and then sets the *preempt_flag*. With Lazy Preemption a preempting interrupt only needs to set the flag notifying the current running thread that it is time to preempt. If there is no thread running, then it is harmless to set the flag, so additional checks in each interrupt are not required, as they are in traditional TOSThreads. In TOSThreads since the main application (which runs the tasks and events) is a thread, they leave the preemption timer on the entire time there is one additional thread in the running state. In our implementation this is not required

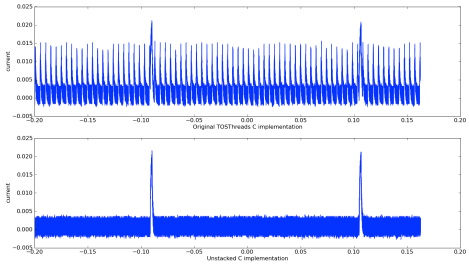


Figure 30: CPU Current Consumption of Blink in TOSThreads vs. `UnStacked C`

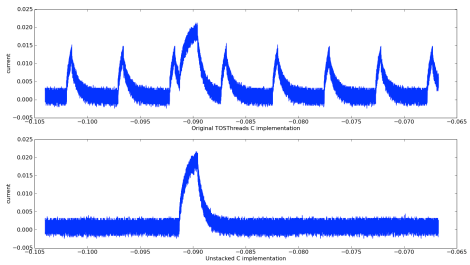


Figure 31: CPU Current During Thread Execution Consumption in TOSThreads vs. `UnStacked C`

since the main system is not a thread, it only runs when two or more application threads are actually for contending for processor time.

Power

When comparing TOSThreads and `UnStacked C with LP` we found that these different ways of implementing preemption have distinct impacts on system power. Since the applications being executed are sensornet applications which may run on batteries for extended amounts of time, power consumption of the applications is absolutely critical. Common sense tells us that `UnStacked C with LP` should consume slightly more power due to the processing overhead that we measured in Section 6.3.1. The results turned out different than expected.

We measured the power consumption of a T-Mote Sky mote, which contains a MSP430F1611 microcontroller [53]. We modified the Blink application, which normally blinks three light emitting diodes (LEDs) at different rates. We changed it so that instead of blinking the LEDs it kept turning them off. This allows the code generation to be identical except for the three instructions which normally turn the LEDs on, now turn the LEDs off.

We measured this test program on the same board, first running the unmodified TOSThreads and then again using `UnStacked C with LP`, shown in Figure 30.

The program is idle for 200 ms, then it wakes up and changes the LEDs, then goes back to sleep. In the TOSThreads implementation we see extra interrupts firing constantly, due to the preemption timer firing. This causes the system to wake up, find that there is no thread running and then exit. A detailed view can be seen in Figure 31. When using `UnStacked C with LP`, there is less interrupt overhead[4], then there is with original TOSThreads.

In total, `UnStacked C with LP` reduces the processor utilization from TOSThreads 11.58% down to 0.68%. This means that the same application compiled with `UnStacked C with LP` can run more than fifteen times longer than one that was compiled with only TOSThreads. This change is minor compared to the change in memory usage.

Memory Usage

Since memory usage reduction is a major motivator for `UnStacked C with LP`, we evaluated the memory utilization of seven existing programs for TOSThreads. We recompiled these applications using TOSThreads alone, and then again with `UnStacked C with LP`. These applications were not modified prior to compiling.

`UnStacked C with LP` uses significantly less RAM than regular TOSThreads as shown in Figure 32. In all of these applications, `UnStacked C with LP` yields a smaller RAM footprint. The reduction in RAM is 35% on average. This reduction in memory is proportional to the amount of memory which was allocated for the thread stacks.

In Figure 32 we separated the thread stacks from the base application to try to show the lower limit of memory savings that could possibly occur. The base application does not change between the TOSThreads and the `UnStacked C with LP`, only the thread stack space changes. On average, the original stack usage of the

[4]Interrupt overhead is the amount of processor time that is taken up by interrupts themselves. Since TOSThreads has a larger postamble on their interrupts, it will take longer to execute those interrupts, taking more processor time, thereby increasing the overall interrupt overhead.

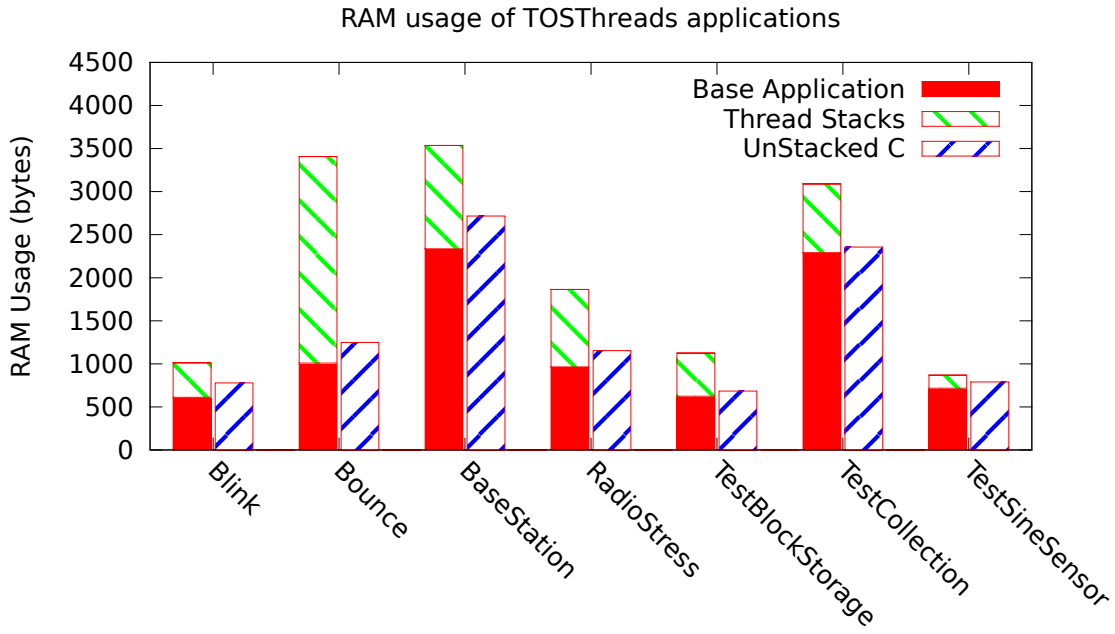


Figure 32: TinyOS RAM usage: TOSThreads vs. `UnStacked C`

original TOSThreads application can be reduced by more than 80% on average.

There are two main reasons why there is more savings in some applications and less in others. The first reason is that there are different numbers of threads in these applications (1-6 threads). The second reason is that the developers of these applications all picked different stack sizes. The number of threads and the individual stack sizes are shown in Table V. They range in values 200 bytes per threads all the way up to 800 bytes per thread. The variety in these stack sizes stems from the fact these stack sizes were educated guesses. This means that there may be a chance of stack overflow in these applications.

This memory reduction comes at a cost. Figure 33 shows how the ROM (flash) grows an average of 12%. This includes the overhead from the normal `UnStacked C` transformation as well as the Lazy Preemption transformation. It also removes the processor specific stack swapping routines and instead uses custom swapping routines for each thread.

Program Name	Number of Threads	RAM per Stack (bytes)
Blink	4	200
Bounce	4	600
BaseStation	6	200
RadioStress	3	300
TestBlockStorage	1	500
TestCollection	1	800

Table V: The number of threads and the size of the stacks define in TOSThreads in each program.

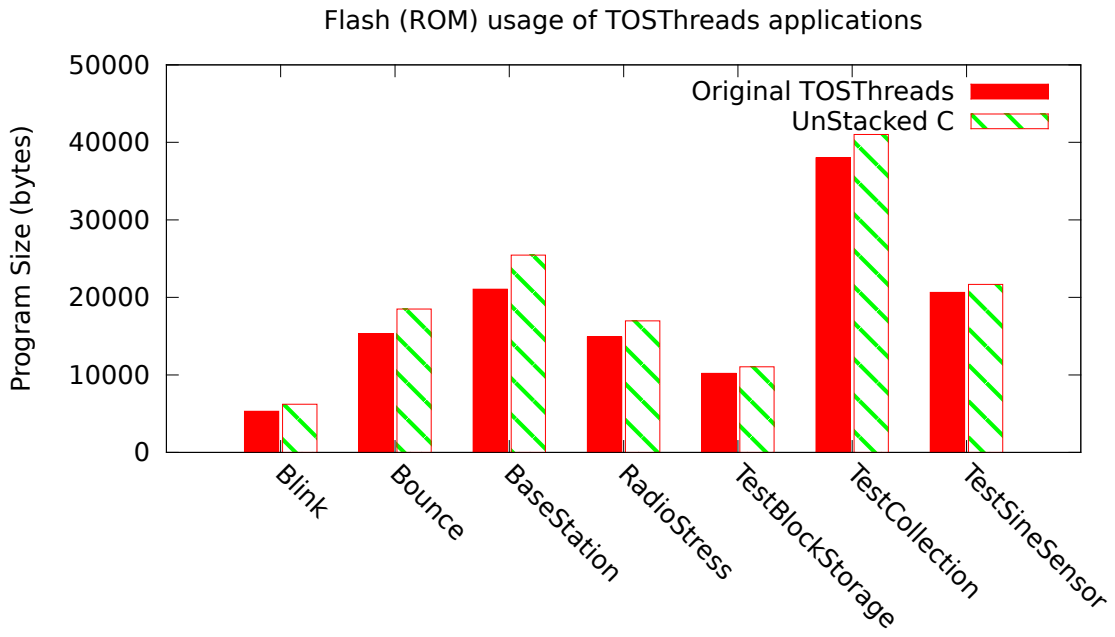


Figure 33: TinyOS Flash usage: TOSThreads vs. `UnStacked C`

The overhead from flash is expected. There is significant code generated in C. We already know that the injected yield points can create an overhead of up to 25% from the simple benchmarks. One of the reasons that the overall program size has not increased to 25% is because only the threaded source code increases and these applications are threaded/event-driven hybrids. Another reason is that these programs are doing more things without loops, so the instruction to optional preemption point ratio is significantly lower than in the microbenchmark.

6.4 Limitations

The limitations from regular `UnStacked C` still exist when `UnStacked C with LP`. These are described in depth in Chapter 5 in Section 5.3.2. There are no additional limitations added from Lazy Preemption.

6.5 Summary

Lazy Preemption is a novel preemption technique. It translates preemptive threads into cooperative threads by injecting optional preemption points throughout the code. When the scheduler wants to preempt the current running thread, it sets a flag, and when the current running thread hits an optional preemption point it yields.

We then extended `UnStacked C with LP` in a C-XML-C transform. This enables existing applications to be recompiled into stackless preemptive multithreads.

We evaluated these optional preemption points in a simple application. We found the processor overhead is insignificant, less than one percent. The program space utilization is significant, at 25%.

`UnStacked C with LP` can recompile existing applications with some minor changes to the scheduler. We evaluated the power consumption in real world applications, finding `UnStacked C with LP` to use fifteen times less power than the previously existing applications.

We used `UnStacked C with LP` to recompile seven existing example applications reducing the overall memory consumption by 35% on average. The memory required for the threads themselves is reduced by more than 80%. The program space utilizations is increased by only 12% in real-world embedded applications. It achieves all of this without a complex stack analysis required which knows all of the interrupt complexities of the rest of the system.

Beyond all of these values, it reduces safety risks in embedded systems, since

thread stack overflows are now impossible. It also enables developers to understand how much memory each thread uses, allowing them to plan larger programs accordingly. This enables embedded developers to use preemptive multithreads with the same ubiquity as PC developers without creating new risks.

CHAPTER VII

Conclusion

Writing software for embedded systems is difficult. We provide several new tools aimed at improving the productivity of embedded system developers. We enable the use of cooperative and preemptive multithreading without worrying about stack allocations or stack overflows. We allow developers to remove indirection and provide a seamless way to implement other compiler-level optimizations. We provide developers a new mechanism for building compilers.

All of these improvements come with the ability to recompile existing applications. This means that legacy applications can be rebuilt to target smaller, lower cost hardware than originally intended. It also can allow new features to be added to existing systems using multithreading without the cost.

7.1 Thesis

We set out to defend the following thesis:

1. A flexible whole-program compiler framework can enable building compilers that can detect and prevent certain faults, or perform optimizations, in embed-

ded system software at compile time.

2. Cooperative multithreads can be translated by a compiler into event-driven state machines. These state machines could have lower memory requirements known at compile time, preventing thread stack overflows.

3. Preemptive multithreads can be translated into similar event-driven state machines with the same bounding of memory requirements.

These new tools can only work when programs are compiled using whole-program compilers. These whole program compilers require all source code for a given application to be compiled together simultaneously. Whole-program compilers can be built from existing compilers or compiler frameworks, but those all required the developers to learn a new language or at least understand an enormous body of code written by someone else. Instead we provide a source to source compiler framework enabling developers to perform analysis or transformations on C source code in their language of choice. Our framework, called C-XML-C, satisfies the first claim of the dissertation.

The flexibility of C-XML-C makes compiler writing more accessible to programmers at large, and not just the few who know compiler internals. We have demonstrated this flexibility by writing eight different transforms in four different programming languages. C-XML-C is a flexible whole-program compiler framework we use to build compilers. Some of these compilers (we call transforms) can detect and prevent faults in thread-safety, data concurrency (Section 4.3.3) and stack overflow (Chapters 5 and 6) at compile time. Other compilers perform optimizations: tail recursion (section 4.3.2), indirection removal (Section 4.3.7) and function inlining (Section 4.3.4). All of these compilers are focused on embedded system developer needs.

Some of these transforms improve system safety, while others allow developers access new system optimizations. All of these reduce the complexity that developers face while designing software components for embedded systems. The most important of these transforms is the translation of multithreaded programs into event-driven state machines.

The translation of multithreaded programs into event-driven state machines has been proposed and implemented in the past in limited circumstances. We provide a way to compile arbitrary cooperative multithreaded C code into event-driven state machines. Our transform that implements this is called `UnStacked C`. `UnStacked C` satisfies the second claim of the thesis. Stackless threads means that stacks need not be allocated and that thread stack overflow is impossible. Prior attempts at stackless multithreading simply disallow local variables and nested function calls, making these approaches unusable for legacy code. We recognize that programs use local variables and nested functions, and that these require some storage location. `UnStacked C` translates the source code to reference a context for each thread, and allows developers to allocate the exact required context for each thread.

We have successfully implemented `UnStacked C` as a C-XML-C transform and evaluated it against a multithreaded system for embedded systems named TinyThread. TinyThread has the lowest stack consumption of any multithreading system since it uses stack analysis to allocate only the theoretical minimum stack space via stack analysis. We recompiled existing TinyThread programs and reduce the overall memory requirements by more than 25% beyond the theoretical limit. All of this is accomplished without creating any risk of stack overflow.

We also ran a microbenchmark on a PC using `UnStacked C`. We were able to achieve over one million threads in less than 50 MB of RAM. While this is out of the focus of our primary research in this dissertation, it leads to new research directions using previously ignored design strategies.

`UnStacked C` also fixes possible faults in data sharing between two cooperative threads. Since variables and registers are not pushed onto a physical stack, temporal data faults cannot occur. This is due to the fact that the threads are no longer threads, but are event-driven state machines, which run one at a time. Since the functions themselves will return, all of the variables must be written before exiting. This means that they will not be caught in registers during a task switch — and cannot create a temporal fault.

We have proposed a new type of preemption called Lazy Preemption. It forces preemptions to only occur at well-defined points in a program's execution. It is implemented by injecting these optional preemption points at the end of each loop iteration. The interrupt which used to force a task switch now sets a flag. This flag is checked at these optional preemption points, and if it is set than the thread will yield. This effectively translates preemptive threads into cooperative threads, but with more yield points to enable preemption.

We then translate the entire preemptive multithreaded application into an event-driven state machine. We implemented Lazy Preemption inside of `UnStacked C`. `UnStacked C with LP` satisfies the final claim of the thesis. This enables existing preemptive multithreaded applications to be recompiled into event-driven state machines. Once translated, these inherit the same safety as other `UnStacked C` applications, in that temporal data faults cannot occur. It is also important to note that the lazy transformation removes the possibility of multi-word data faults, since it only yields in between lines of source code. This allows a given line to complete executing regardless of how many instructions it takes.

We tested `UnStacked C with LP` by recompiling existing applications initially written in a preemptive multithreading system called TOSThreads. We measured the power consumption in one of these applications and found that `UnStacked C with LP` uses 15 times less power. The memory used in these applications is re-

duced by 35% on average. It reduces the memory required for the given threads by more than 80%.

`UnStacked C` reduces RAM usage at a cost of an increase in program space. When we measured `UnStacked C` against TinyThread we saw an increase in program space of 6.9% on average. When we measured `UnStacked C with LP` on TOSThread applications we saw an increase of program space of 12%. We see these costs as nominal compared to the reduction in memory usage. All of these applications only have a single instance of every thread, in systems that have multiple instances of each thread, only the number of contexts increase, not the program size. This means that other applications may have a more significant benefit with a smaller cost.

The benefits to embedded systems is clear: Multithreading in embedded systems is now possible without the traditional risks of multithreading at only a nominal cost. This enables developers to write applications without such an in-depth knowledge of multithreading systems. It also allows them to write larger applications than the RAM would typically allow.

7.2 Future Work

We view C-XML-C as a technology that will enable other research. It lowers the barrier to manipulating C compilers, allowing many different developers to look at problems differently. We initially built it so that we could build the complex transformations required to perform `UnStacked C`. It took on a life of its own when we realized its flexibility and ease of use. In this regard, C-XML-C may enable a large body of research outside of the scope of embedded systems.

We view `UnStacked C` in a similar light. We view `UnStacked C` as a foundation for many different types of multithreading systems to come. In the embedded

systems world, real-time operating system (RTOS) vendors are constantly looking to out perform each other, and `UnStacked C` is one such approach. If `UnStacked C with LP` is used by one such RTOS, it is likely that others will follow to remain competitive.

Outside of the embedded systems world, `UnStacked C` can lay the foundation for more complex threading system on PCs and servers. High-concurrency servers often require many threads and are limited by the constraints of the overhead of threads and existing design patterns. `UnStacked C` could be extended to relieve the constraints of multithreading overhead and increase performance of existing systems.

.

BIBLIOGRAPHY

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference, General Track*, pages 289–302, 2002.

[2] G. Antoniol, M. D. Penta, G. Masone, and U. Villano. Xogastan: Xml-oriented gcc ast analysis and transformations. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:173, 2003.

[3] V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, and L. Wood. Document Object Model (DOM) Level 1 Specification (Second Edition). `http://www.w3.org/TR/2000/WD-DOM-Level-1-20000929/` Version 1.0. W3C Working Draft 29 September, 2000, Sept. 2000.

[4] ATMEL Corporation. 8-bit avr microcontroller with 4/8/16k bytes in-system programmable flash. `http://www.atmel.com/dyn/resources/prod_documents/doc2545.pdf`, 2010.

[5] ATMEL Corporation. At91sam arm-based flash mcu: Same3s series preliminary. `http://www.atmel.com/dyn/resources/prod_documents/doc6500.pdf`, 2011.

[6] L. L. Beck. *System Software: An Introduction to Systems Programming (3rd Edition)*. Addison Wesley Longman, Inc., Boston, MA, USA, 1997.

[7] M. Bergsma, M. Holenderski, R. J. Bril, and J. J. Lukkien. Extending rtai/linux with fixed-priority scheduling with deferred preemption. In *5th International*

Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2009), Dublin, Ireland, June 2009.

[8] A. Bernauer, K. Römer, and S. Santini. Threads for the programmer, events for the machine. In *Adjunct Proceedings of the 7th European Conference on Wireless Sensor Networks (EWSN 2010)*, pages 77–79, Coimbra, Portugal, Feb. 2010.

[9] A. Bernauer, K. Römer, S. Santini, and J. Ma. Threads2events: An automatic code generation approach. In *Proceedings of the 6th Workshop on Hot Topics in Embedded Networked Sensors (HotEMNETS 2010)*, Killarney, Ireland, June 2010.

[10] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. *Journal of Parallel and Distributed Computing*, 37(1):55–69, August 25 1996. (An early version appeared in the *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '95)*, pages 207–216, Santa Barbara, California, July 1995.).

[11] R. Bril, J. Lukkien, and W. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Systems*, 42:63–119, 2009. 10.1007/s11241-009-9071-z.

[12] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *Proc. of the 23rd Intl. Conf. on Software Engineering (ICSE)*, pages 47–56. ACM/IEEE, 2001.

[13] A. Burns. Preemptive priority based scheduling: An appropriate engineering approach. In *Principles of Real-time Systems*, pages 225–248. Prentice Hall, 1994.

[14] A. Burns, M. Nicholson, K. Tindell, and N. Zhang. Allocating and scheduling hard real-time tasks on a point-to-point distributed system. In *Proceedings of the Workshop on Parallel and Distributed Real-Time Systems*, pages 11–20, 1993.

[15] W. D. Clinger. Proper tail recursion and space efficiency. *SIGPLAN Not.*, 33:174–185, May 1998.

[16] N. Cooprider and J. Regehr. Pluggable abstract domains for analyzing embedded software. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED conference on Language, compilers, and tool support for embedded systems*, LCTES '06, pages 44–53, New York, NY, USA, 2006. ACM.

[17] T. R. Dean, A. J. Malton, and R. Holt. Union schemas as a basis for a c++ extractor. *Reverse Engineering, Working Conference on*, 0:59, 2001.

[18] T. Duff. Duff's device. `http://www.lysator.liu.se/c/duffs-device.html`.

[19] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[20] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM.

[21] Edward L. Lamie. *Real-Time Embedded Multithreading Using ThreadX*. Elsevier Inc., 2009.

[22] Express Logic, Inc. Helping you avoid stack overflow crashes! http://rtos.com/images/uploads/Stack_Analysis_White_paper.1_.pdf.

[23] I. Free Software Foundation. Alloca(3) linux programmer's manual. http://www.kernel.org/doc/man-pages/online/pages/man3/alloca.3.html, 2008.

[24] I. Free Software Foundation. GCC RUNTIME LIBRARY EXCEPTION. http://www.gnu.org/licenses/gcc-exception.html, 2009.

[25] I. Free Software Foundation. Malloc(3) linux programmer's manual. http://www.kernel.org/doc/man-pages/online/pages/man3/malloc.3.html, 2010.

[26] V. W. Freeh, D. K. Lowenthal, and G. R. Andrews. Distributed filaments: efficient fine-grain parallelism on a cluster of workstations. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 15, Berkeley, CA, USA, 1994. USENIX Association.

[27] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation*, pages 212–223, Montreal, Quebec, Canada, June 1998. Proceedings published ACM SIGPLAN Notices, Vol. 33, No. 5, May, 1998.

[28] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[29] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03:*

Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, pages 1–11, New York, NY, USA, 2003. ACM Press.

[30] P. Gootherts and D. Larson. Memory mapped lazy preemption control, November 2006.

[31] T. Gschwind, M. Pinzger, and H. Gall. Tuanalyzer " analyzing templates in c++ code. In *WCRE '04: Proceedings of the 11th Working Conference on Reverse Engineering*, pages 48–57, Washington, DC, USA, 2004. IEEE Computer Society.

[32] B. Gu, Y. Kim, J. Heo, and Y. Cho. Shared-stack cooperative threads. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1181–1186, New York, NY, USA, 2007. ACM.

[33] L. Gu. t-kernel: Providing reliable os support to wireless sensor networks. In *In Proc. of the 4th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, pages 1–14. ACM Press, 2006.

[34] A. Gustafsson. Threads without the pain. *Queue*, 3(9):34–41, 2005.

[35] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS-IX: Proceedings of the ninth international conference on architectural support for programming languages and operating systems*, pages 93–104, New York, NY, USA, 2000. ACM Press.

[36] M. Holenderski, R. J. Bril, and J. J. Lukkien. Using fixed priority scheduling with deferred preemption to exploit fluctuating network bandwidth. In *ECRTS*, pages 40–43, 2008.

[37] R. C. Holt, A. Winter, and A. Schrr. Gxl: Toward a standard exchange format. *Reverse Engineering, Working Conference on*, 0:162, 2000.

[38] N. Instruments. Labview 8.5.x known issues. `http://zone.ni.com/devzone/cda/tut/p/id/6449%5C#51084_by_Category`, 2009.

[39] ISO. Iso c standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[40] J. Clark. Xsl transformations (xslt)version 1.0. `http://www.w3.org/TR/xslt`, 1999.

[41] Jack Ganssle. *The Art of Designing Embedded Systems*. Butterworth-Heinemann, 2000.

[42] Jakob Bernoulli. *Ars Conjectandi*. 1713.

[43] Jean J. Labrosse. *uC/OS-III The Real-Time Kernel*. Micrium Press, 2010.

[44] K. Klues, C.-J. M. Liang, J. Paek, R. Musăloiu-E, P. Levis, A. Terzis, and R. Govindan. Tosthreads: thread-safe and non-invasive preemption in tinyos. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 127–140, New York, NY, USA, 2009. ACM.

[45] M. Krohn, E. Kohler, and M. F. Kaashoek. Events can make sense. In *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX 2007)*, Santa Clara, CA, June 2007.

[46] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[47] N. R. Laurent George and M. Spuri. Preemptive and nonpreemptive real-time uniprocessor scheduling. Technical Report RR-2966, Inria, Institut National de Recherche en Informatique et en Automatique, 1996.

[48] C.-G. Lee, J. Hahn, Y.-M. Seo, S. L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47:700–713, June 1998.

[49] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20:46–61, January 1973.

[50] W. P. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 167–180, New York, NY, USA, 2006. ACM.

[51] J. A. Meister, J. S. Foster, and M. Hicks. Serializing C intermediate representations for efficient and portable parsing. *Software, Practice, and Experience*, 40(3):225–238, Feb. 2010.

[52] A. Milanova, A. Rountev, and B. G. Ryder. Precise call graphs for c programs with function pointers. *Automated Software Engg.*, 11:7–26, January 2004.

[53] Moteiv Corporation. Tmote Sky data sheet. `http://moteiv.com/products/docs/tmote-sky-datasheet.pdf`, 2005.

[54] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[55] Niclas Lindblom. Controlling the stack size. `http://www.iar.com/website1/1.0.1.0/483/1/`.

[56] C. Nitta, R. Pandey, and Y. Ramin. Y-threads: Supporting concurrency in wireless sensor networks. In *DCOSS*, pages 169–184, 2006.

[57] S. Oualline. *Vi IMproved VIM*. New Riders, 2001.

[58] D. Patterson and J. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 2005.

[59] B. C. Peter Roses, Basile Starynkevitch. GCC Plugins. `http://gcc.gnu.org/wiki/GCC_Plugins`, 2008.

[60] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemption points. In *Proceedings of the 27th IEEE International Real-Time Systems Symposium*, pages 212–224, Washington, DC, USA, 2006. IEEE Computer Society.

[61] H. Ramaprasad and F. Mueller. Tightening the bounds on feasible preemptions. *ACM Trans. Embed. Comput. Syst.*, 10:27:1–27:34, January 2011.

[62] J. Regehr. Random testing of interrupt-driven software. In *Proceedings of the 5th ACM international conference on Embedded software*, EMSOFT '05, pages 290–298, New York, NY, USA, 2005. ACM.

[63] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *Trans. on Embedded Computing Sys.*, 4(4):751–778, 2005.

[64] Richard Barry. Stack Usage and Stack Overflow Checking. `http://www.freertos.org/Stacks-and-stack-overflow-checking.html`.

[65] J. Sallai, M. Maroti, and A. Ledeczi. A concurrency abstraction for reliable sensor network applications. In F. Kordon and J. Sztipanovits, editors, *Reliable Systems on Unreliable Networked Platforms*, volume 4322 of *Lecture Notes in Computer Science*, pages 143–160. Springer Berlin / Heidelberg, 2007.

[66] A. S. Tanenbaum. *Modern operating systems*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[67] T. G. Team. The gcc c-torture test suite. `http://gcc.gnu.org/install/test.html`.

[68] T. G. Team. The gnu compiler collection. `http://gcc.gnu.org/`.

[69] Texas Instruments Incorporated. The msp430 hardware multiplier: Functions and applications. `http://focus.ti.com/lit/an/slaa042/slaa042.pdf`, 1999.

[70] The Eclipse Foundation. Eclipse. `http://www.eclipse.org/`.

[71] C. Tismer. Continuations and stackless python or "how to change a paradigm of an existing program". In *Proceedings of the 8th International Python Conference*, 2000.

[72] G. van Rossum. Python frequently asked questions. `http://docs.python.org/download.html`, 2010.

[73] E. Visser. Stratego: A language for program transformation based on rewriting strategies system description of stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer Berlin / Heidelberg, 2001.

[74] J. R. von Behren, J. Condit, and E. A. Brewer. Why events are a bad idea (for high-concurrency servers). In M. B. Jones, editor, *HotOS*, pages 19–24. USENIX, 2003.

[75] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 268–281, New York, NY, USA, 2003. ACM Press.

[76] M. Welsh, D. Culler, and E. Brewer. Seda: an architecture for well-conditioned, scalable internet services. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 230–243, New York, NY, USA, 2001. ACM.

[77] Werner Boellmann, Dean Camera, Pieter Conradie, Brian Dean,Keith Gudger, Wouter van Gulik, Bjoern Haase, Steinar Haugen, Peter Jansen, Reinhard Jessich, Magnus Johansson, Harald Kipp, Carlos Lamas, Cliff Lawson, Artur Lipowski, Marek Michalkiewicz, Todd C. Miller, Rich Neswold, Colin O'Flynn, Bob Paddock, Andrey Pashchenko, Reiner Patommel, Florin-Viorel Petrov, Alexander Popov, Michael Rickman, Theodore A. Roth, Juergen Schilling, Philip Soeberg, Anatoly Sokolov, Nils Kristian Strom, Michael Stumpf, Stefan Swanepoel, Helmut Wallner, Eric B. Weddington, Joerg Wunsch, Dmitry Xmelkov, Atmel Corporation, egnite Software GmbH, The Regents of the University of California. Avr-libc users manual. `http://www.nongnu.org/avr-libc/user-manual/FAQ.html`.

[78] X. Yang, N. Cooprider, and J. Regehr. Eliminating the call stack to save ram. In *Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '09, pages 60–69, New York, NY, USA, 2009. ACM.

[79] G. Yao, G. Buttazzo, and M. Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, RTCSA '09, pages 351–360, Washington, DC, USA, 2009. IEEE Computer Society.

[80] J. Yi, C. Sadowski, and C. Flanagan. Cooperative reasoning for preemptive execution. In *Proceedings of the 16th ACM symposium on Principles and practice*

of parallel programming, PPoPP '11, pages 147–156, New York, NY, USA, 2011. ACM.

[81] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: a step toward practical analyses. *SIGSOFT Softw. Eng. Notes*, 21:81–92, October 1996.